www.sapientia.uji.es | 29



Introducción a la programación con C

Andrés Marzal

Isabel Gracia

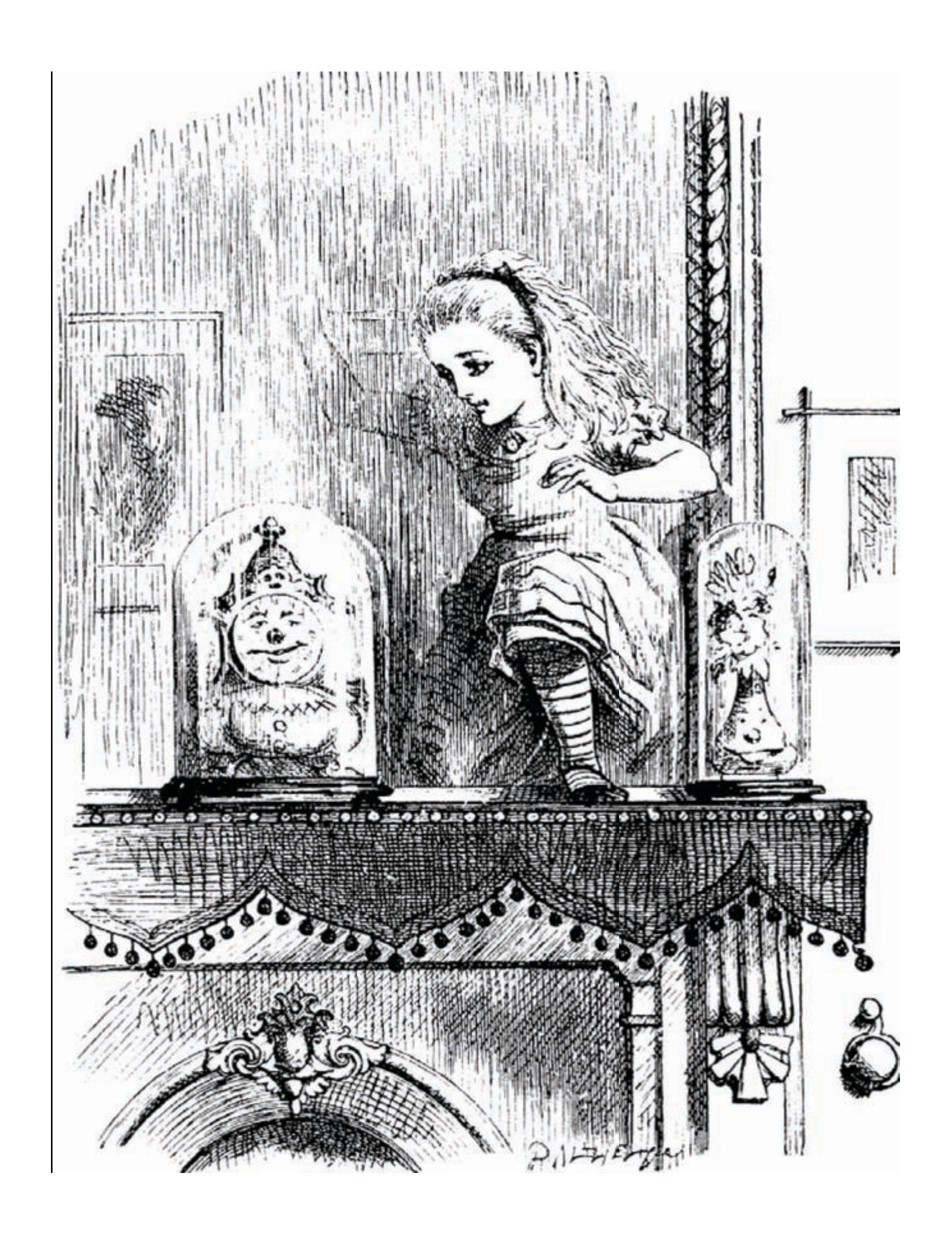
Introducción a la programación con C

Andrés Marzal
Isabel Gracia



Departamento de lenguajes y sistemas informáticos

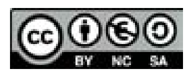
■ Codis d'assignatura II04 i IG04



Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Col·lecció Sapientia, 29
www.sapientia.uji.es

ISBN: 978-84-693-0143-2



Índice general

1. Introducción a C 1
1.1. C es un lenguaje compilado . . . 3
1.2. Traduciendo de Python a C: una guía rápida . . . 5
1.3. Estructura típica de un programa C . . . 14
1.4. C es un lenguaje de formato libre . . . 15
1.5. Hay dos tipos de comentario . . . 21
1.6. Valores literales en C . . . 24
1.6.1. Enteros . . . 24
1.6.2. Flotantes . . . 24
1.6.3. Cadenas . . . 24
1.7. C tiene un rico juego de tipos escalares . . . 25
1.7.1. El tipo **int** . . . 26
1.7.2. El tipo **unsigned int** . . . 26
1.7.3. El tipo **float** . . . 26
1.7.4. El tipo **char** . . . 27
1.7.5. El tipo **unsigned char** . . . 27
1.8. Se debe declarar el tipo de toda variable antes de usarla . . . 27
1.8.1. Identificadores válidos . . . 27
1.8.2. Sentencias de declaración . . . 27
1.8.3. Declaración con inicialización . . . 29
1.9. Salida por pantalla . . . 30
1.9.1. Marcas de formato para la impresión de valores con *printf* . . . 30
1.10. Variables y direcciones de memoria . . . 35
1.11. Entrada por teclado . . . 38
1.12. Expresiones . . . 38
1.13. Conversión implícita y explícita de tipos . . . 46
1.14. Las directivas y el preprocesador . . . 49
1.15. Constantes . . . 49
1.15.1. Definidas con la directiva *define* . . . 49
1.15.2. Definidas con el adjetivo **const** . . . 50
1.15.3. Con tipos enumerados . . . 50
1.16. Las bibliotecas (módulos) se importan con **#include** . . . 53
1.16.1. La biblioteca matemática . . . 53
1.17. Estructuras de control . . . 55
1.17.1. Estructuras de control condicionales . . . 55
1.17.2. Estructuras de control iterativas . . . 60
1.17.3. Sentencias para alterar el flujo iterativo . . . 67

2. **Estructuras de datos en C: vectores estáticos y registros** **69**
2.1. Vectores estáticos 69
2.1.1. Declaración de vectores 69
2.1.2. Inicialización de los vectores 70
2.1.3. Un programa de ejemplo: la criba de Eratóstenes 73
2.1.4. Otro programa de ejemplo: estadísticas 74
2.1.5. Otro programa de ejemplo: una calculadora para polinomios 85
2.1.6. Disposición de los vectores en memoria 91
2.1.7. Algunos problemas de C: accesos ilícitos a memoria 95
2.1.8. Asignación y copia de vectores 97
2.1.9. Comparación de vectores 99
2.2. Cadenas estáticas 99
2.2.1. Declaración de cadenas 100
2.2.2. Representación de las cadenas en memoria 100
2.2.3. Entrada/salida de cadenas 102
2.2.4. Asignación y copia de cadenas 107
2.2.5. Longitud de una cadena 111
2.2.6. Concatenación 115
2.2.7. Comparación de cadenas 116
2.2.8. Funciones útiles para manejar caracteres 117
2.2.9. Escritura en cadenas: *sprintf* 118
2.2.10. Un programa de ejemplo 119
2.3. Vectores multidimensionales 120
2.3.1. Sobre la disposición de los vectores multidimensionales en memoria 120
2.3.2. Un ejemplo: cálculo matricial 121
2.3.3. Vectores de cadenas, matrices de caracteres 128
2.4. Registros 134
2.4.1. Un ejemplo: registros para almacenar vectores de talla variable (pero acotada) 139
2.4.2. Un ejemplo: rectas de regresión para una serie de puntos en el plano 141
2.4.3. Otro ejemplo: gestión de una coleción de CDs 144
2.5. Definición de nuevos tipos de datos 149

3. **Funciones** **152**
3.1. Definición de funciones 152
3.2. Variables locales y globales 157
3.2.1. Variables locales 157
3.2.2. Variables globales 159
3.3. Funciones sin parámetros 160
3.4. Procedimientos 162
3.5. Paso de parámetros 163
3.5.1. Parámetros escalares: paso por valor 163
3.5.2. Organización de la memoria: la pila de llamadas a función 163
3.5.3. Vectores de longitud variable 169
3.5.4. Parámetros vectoriales: paso por referencia 170
3.5.5. Parámetros escalares: paso por referencia mediante punteros 177
3.5.6. Paso de registros a funciones 182
3.5.7. Paso de matrices y otros vectores multidimensionales 185
3.5.8. Tipos de retorno válidos 189
3.5.9. Un ejercicio práctico: miniGalaxis 190
3.6. Recursión 209

3.6.1. Un método recursivo de ordenación: mergesort 210
3.6.2. Recursión indirecta y declaración anticipada 216
3.7. Macros 217
3.8. Otras cuestiones acerca de las funciones 220
3.8.1. Funciones **inline** 220
3.8.2. Variables locales **static** 221
3.8.3. Paso de funciones como parámetros 222
3.9. Módulos, bibliotecas y unidades de compilación 224
3.9.1. Declaración de prototipos en cabeceras 227
3.9.2. Declaración de variables en cabeceras 229
3.9.3. Declaración de registros en cabeceras 229

4. Estructuras de datos: memoria dinámica **234**
4.1. Vectores dinámicos 234
4.1.1. *malloc*, *free* y `NULL` 235
4.1.2. Algunos ejemplos 238
4.1.3. Cadenas dinámicas 252
4.2. Matrices dinámicas 254
4.2.1. Gestión de memoria para matrices dinámicas 254
4.2.2. Definición de un tipo «matriz dinámica» y de funciones para su gestión 258
4.3. Más allá de las matrices dinámicas 261
4.3.1. Vectores de vectores de tallas arbitrarias 261
4.3.2. Arreglos *n*-dimensionales 266
4.4. Redimensionamiento de la reserva de memoria 268
4.4.1. Una aplicación: una agenda telefónica 278
4.5. Introducción a la gestión de registros enlazados 283
4.5.1. Definición y creación de la lista 285
4.5.2. Adición de nodos (por cabeza) 286
4.5.3. Adición de un nodo (por cola) 289
4.5.4. Borrado de la cabeza 292
4.6. Listas con enlace simple 294
4.6.1. Creación de lista vacía 295
4.6.2. ¿Lista vacía? 296
4.6.3. Inserción por cabeza 296
4.6.4. Longitud de una lista 299
4.6.5. Impresión en pantalla 300
4.6.6. Inserción por cola 301
4.6.7. Borrado de la cabeza 303
4.6.8. Borrado de la cola 304
4.6.9. Búsqueda de un elemento 308
4.6.10. Borrado del primer nodo con un valor determinado 309
4.6.11. Borrado de todos los nodos con un valor dado 311
4.6.12. Inserción en una posición dada 313
4.6.13. Inserción ordenada 314
4.6.14. Concatenación de dos listas 316
4.6.15. Borrado de la lista completa 316
4.6.16. Juntando las piezas 317
4.7. Listas simples con punteros a cabeza y cola 324
4.7.1. Creación de lista vacía 325
4.7.2. Inserción de nodo en cabeza 326
4.7.3. Inserción de nodo en cola 327

4.7.4. Borrado de la cabeza 328
4.7.5. Borrado de la cola 329
4.8. Listas con enlace doble 331
4.8.1. Inserción por cabeza 331
4.8.2. Borrado de la cola 333
4.8.3. Inserción en una posición determinada 334
4.8.4. Borrado de la primera aparición de un elemento 336
4.9. Listas con enlace doble y puntero a cabeza y cola 339
4.10. Una guía para elegir listas 341
4.11. Una aplicación: una base de datos para discos compactos 343
4.12. Otras estructuras de datos con registros enlazados 350

5. Ficheros **353**
5.1. Ficheros de texto y ficheros binarios 353
5.1.1. Representación de la información en los ficheros de texto 353
5.1.2. Representación de la información en los ficheros binarios 354
5.2. Ficheros de texto 355
5.2.1. Abrir, leer/escribir, cerrar 355
5.2.2. Aplicaciones: una agenda y un gestor de una colección de discos compactos 363
5.2.3. Los «ficheros» de consola 374
5.2.4. Un par de utilidades 377
5.3. Ficheros binarios 378
5.3.1. Abrir, leer/escribir, cerrar 378
5.3.2. Acceso directo 384
5.4. Errores 389

A. Tipos básicos **391**
A.1. Enteros 391
A.1.1. Tipos 391
A.1.2. Literales 392
A.1.3. Marcas de formato 392
A.2. Flotantes 393
A.2.1. Tipos 393
A.2.2. Literales 393
A.2.3. Marcas de formato 393
A.3. Caracteres 394
A.3.1. Literales 394
A.3.2. Marcas de formato 394
A.4. Otros tipos básicos 394
A.4.1. El tipo booleano 394
A.4.2. Los tipos complejo e imaginario 394
A.5. Una reflexión acerca de la diversidad de tipos escalares 395

B. La lectura de datos por teclado, paso a paso **396**
B.1. La lectura de valores escalares con *scanf* 396
B.2. La lectura de cadenas con *scanf* 400
B.3. Un problema serio: la lectura alterna de cadenas con *gets* y de escalares con *scanf* 403

Capítulo 1

Introducción a C

> *Había un libro junto a Alicia, en la mesa; y mientras permanecía sentada observando al Rey Blanco [. . .], pasaba las hojas para ver si encontraba algún trozo que poder leer: «. . . Porque está todo en una lengua que no entiendo», se dijo.*
> *Estaba así:*
>
> JERIGÓNDOR
>
> Cocillaba el día y las tovas agilimosas
> giroscopaban y barrenaban en el larde.
> Todo devirables estaban los burgovos,
> y silbramaban las alecas rastas.
>
> *Durante un rato, estuvo contemplando esto perpleja; pero al final se le ocurrió una brillante idea. ¡Ah, ya sé!, ¡es un libro del Espejo, naturalmente! Si lo pongo delante de un espejo, las palabras se verán otra vez del derecho.*
>
> LEWIS CARROLL, *Alicia a través del espejo.*

El lenguaje de programación C es uno de los más utilizados (si no el que más) en la programación de sistemas software. Es similar a Python en muchos aspectos fundamentales: presenta las mismas estructuras de control (selección condicional, iteración), permite trabajar con algunos tipos de datos similares (enteros, flotantes, secuencias), hace posible definir y usar funciones, etc. No obstante, en muchas otras cuestiones es un lenguaje muy diferente.

C presenta ciertas características que permiten ejercer un elevado control sobre la eficiencia de los programas, tanto en la velocidad de ejecución como en el consumo de memoria, pero a un precio: tenemos que proporcionar información explícita sobre gran cantidad de detalles, por lo que generalmente resultan programas más largos y complicados que sus equivalentes en Python, aumentando así la probabilidad de que cometamos errores.

En este capítulo aprenderemos a realizar programas en C del mismo «nivel» que los que sabíamos escribir en Python tras estudiar el capítulo 4 del primer volumen. Aprenderemos, pues, a usar variables, expresiones, la entrada/salida, funciones definidas en «módulos» (que en C se denominan bibliotecas) y estructuras de control. Lo único que dejamos pendiente de momento es el tratamiento de cadenas en C, que es sensiblemente diferente al que proporciona Python.

Nada mejor que un ejemplo de programa en los dos lenguajes para que te lleves una primera impresión de cuán diferentes son Python y C… y cuán semejantes. Estos dos programas, el primero en Python y el segundo en C, calculan el valor de

$$\sum_{i=a}^{b} \sqrt{i}$$

para sendos valores enteros de a y b introducidos por el usuario y tales que $0 \leq a \leq b$.

sumatorio.py **sumatorio.py**

```python
from math import *

# Pedir límites inferior y superior.
a = int(raw_input('Límite inferior:'))
while a < 0:
   print 'No puede ser negativo'
   a = int(raw_input('Límite inferior:'))

b = int(raw_input('Límite superior:'))
while b < a:
   print 'No puede ser menor que %d' % a
   b = int(raw_input('Límite superior:'))

# Calcular el sumatorio de la raíz cuadrada de i para i entre a y b.
s = 0.0
for i in range(a, b+1):
   s += sqrt(i)

# Mostrar el resultado.
print 'Sumatorio de raíces',
print 'de %d a %d: %f' % (a, b, s)
```

sumatorio.c **sumatorio.c**

```c
#include <stdio.h>
#include <math.h>

int main(void)
{
   int a, b, i;
   float s;

   /* Pedir límites inferior y superior. */
   printf("Límite inferior:");
   scanf("%d", &a);
   while (a < 0) {
      printf("No puede ser negativo\n");
      printf("Límite inferior:");
      scanf("%d", &a);
   }

   printf("Límite superior:");
   scanf("%d", &b);
   while (b < a) {
      printf("No puede ser menor que %d\n", a);
      printf("Límite superior:");
      scanf("%d", &b);
   }
```

```
25
26    /* Calcular el sumatorio de la raíz cuadrada de i para i entre a y b. */
27    s = 0.0;
28    for (i = a; i <= b; i++) {
29      s += sqrt(i);
30    }
31
32    /* Mostrar el resultado. */
33    printf("Sumatorio␣de␣raíces␣");
34    printf("de␣%d␣a␣%d:␣%f\n", a, b, s);
35
36    return 0;
37  }
```

En varios puntos de este capítulo haremos referencia a estos dos programas. No los pierdas de vista.

. . . . EJERCICIOS . . . .

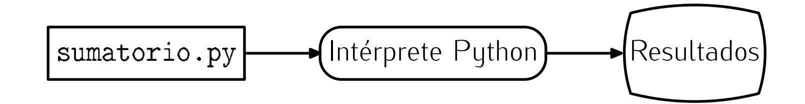
▶ 1 Compara los programas `sumatorio.py` y `sumatorio.c`. Analiza sus semejanzas y diferencias. ¿Qué función desempeñan las llaves en `sumatorio.c`? ¿Qué función crees que desempeñan las líneas 6 y 7 del programa C? ¿A qué elemento de Python se parecen las dos primeras líneas de `sumatorio.c`? ¿Qué similitudes y diferencias aprecias entre las estructuras de control de Python y C? ¿Cómo crees que se interpreta el bucle **for** del programa C? ¿Por qué algunas líneas de `sumatorio.c` finalizan en punto y coma y otras no? ¿Qué diferencias ves entre los comentarios Python y los comentarios C?

. . . .

1.1. C es un lenguaje compilado

Python y C no sólo se diferencian en su sintaxis, también son distintos en el modo en que se traducen los programas a código de máquina y en el modo en que ejecutamos los programas.

- Python es un lenguaje *interpretado*: para ejecutar un programa Python, suministramos al intérprete un fichero de texto (típicamente con extensión «`.py`») con su código fuente. Si deseamos ejecutar `sumatorio.py`, por ejemplo, hemos de escribir `python sumatorio.py` en la línea de órdenes Unix. Como resultado, el intérprete va leyendo y ejecutando paso a paso el programa. Para volver a ejecutarlo, has de volver a escribir `python sumatorio.py` en la línea de órdenes, con lo que se repite el proceso completo de traducción y ejecución paso a paso. Aunque no modifiquemos el código fuente, es necesario interpretarlo (traducir y ejecutar paso a paso) nuevamente.



- C es un lenguaje *compilado*: antes de ejecutar un programa escrito por nosotros, suministramos su código fuente (en un fichero con extensión «`.c`») a un compilador de C. El compilador lee y analiza todo el programa. Si el programa está correctamente escrito según la definición del lenguaje, el compilador genera un nuevo fichero con su traducción a código de máquina, y si no, muestra los errores que ha detectado. Para ejecutar el programa utilizamos el nombre del fichero generado. Si no modificamos el código fuente, no hace falta que lo compilemos nuevamente para

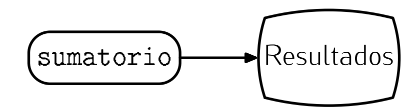
> ***Un poco de historia***
>
> C ya tiene sus añitos. El nacimiento de C está estrechamente vinculado al del sistema operativo Unix. El investigador Ken Thompson, de AT&T, la compañía telefónica estadounidense, se propuso diseñar un nuevo sistema operativo a principios de los setenta. Disponía de un PDP-7 en el que codificó una primera versión de Unix en lenguaje ensamblador. Pronto se impuso la conveniencia de desarrollar el sistema en un lenguaje de programación de alto nivel, pero la escasa memoria del PDP-7 (8K de 18 bits) hizo que ideara el lenguaje de programación B, una versión reducida de un lenguaje ya existente: BCPL. El lenguaje C apareció como un B mejorado, fruto de las demandas impuestas por el desarrollo de Unix. Dennis Ritchie fue el encargado del diseño del lenguaje C y de la implementación de un compilador para él sobre un PDP-11.
>
> C ha sufrido numerosos cambios a lo largo de su historia. La primera versión «estable» del lenguaje data de 1978 y se conoce como «K&R C», es decir, «C de Kernighan y Ritchie». Esta versión fue descrita por sus autores en la primera edición del libro «The C Programming Language» (un auténtico «best-seller» de la informática). La adopción de Unix como sistema operativo de referencia en las universidades en los años 80 popularizó enormemente el lenguaje de programación C. No obstante, C era atractivo por sí mismo y parecía satisfacer una demanda real de los programadores: disponer de un lenguaje de alto nivel con ciertas características propias de los lenguajes de bajo nivel (de ahí que a veces se diga que C es un lenguaje de nivel intermedio).
>
> La experiencia con lenguajes de programación diseñados con anterioridad, como Lisp o Pascal, demuestra que cuando el uso de un lenguaje se extiende es muy probable que proliferen variedades dialectales y extensiones para aplicaciones muy concretas, lo que dificulta enormemente el intercambio de programas entre diferentes grupos de programadores. Para evitar este problema se suele recurrir a la creación de un comité de expertos que define la versión oficial del lenguaje. El comité ANSI X3J9 (ANSI son las siglas del American National Standards Institute), creado en 1983, considera la inclusión de aquellas extensiones y mejoras que juzga de suficiente interés para la comunidad de programadores. El 14 de diciembre de 1989 se acordó qué era el «C estándar» y se publicó el documento con la especificación en la primavera de 1990. El estándar se divulgó con la segunda edición de «The C Programming Language», de Brian Kernighan y Dennis Ritchie. Un comité de la International Standards Office (ISO) ratificó el documento del comité ANSI en 1992, convirtiéndolo así en un estándar internacional. Durante mucho tiempo se conoció a esta versión del lenguaje como ANSI-C para distinguirla así del K&R C. Ahora se prefiere denominar a esta variante C89 (o C90) para distinguirla de la revisión que se publicó en 1999, la que se conoce por C99 y que es la versión estándar de C que estudiaremos.
>
> C ha tenido un gran impacto en el diseño de otros muchos lenguajes. Ha sido, por ejemplo, la base para definir la sintaxis y ciertos aspectos de la semántica de lenguajes tan populares como Java y C++.

volver a ejecutar el programa: basta con volver a ejecutar el fichero generado por el compilador.

Para ejecutar `sumatorio.c`, por ejemplo, primero hemos de usar un compilador para producir un nuevo fichero llamado `sumatorio`.

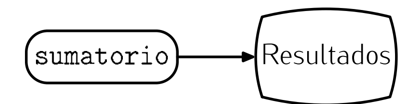


Podemos ejecutar el programa escribiendo `sumatorio` en la línea de órdenes Unix.[1]



[1]Por razones de seguridad es probable que no baste con escribir `sumatorio` para poder ejecutar un programa con ese nombre y que reside en el directorio activo. Si es así, prueba con `./sumatorio`.

Si queremos volver a ejecutarlo, basta con escribir de nuevo `sumatorio`; no es necesario volver a compilar el contenido del fichero `sumatorio.c`.



La principal ventaja de compilar los programas es que se gana en velocidad de ejecución, ya que cuando el programa se ejecuta está completamente traducido a código de máquina y se ahorra el proceso de «traducción simultánea» que conlleva interpretar un programa. Pero, además, como se traduce a código de máquina en una fase independiente de la fase de ejecución, el programa traductor puede dedicar más tiempo a intentar encontrar la mejor traducción posible, la que proporcione el programa de código de máquina más rápido (o que consuma menos memoria).

Nosotros usaremos un compilador concreto de C: `gcc` (en su versión 3.2 o superior)[2]. Su forma de uso más básica es ésta:

gcc *fichero.c* -o *fichero_ejecutable*

La opción `-o` es abreviatura de «output», es decir, «salida», y a ella le sigue el nombre del fichero que contendrá la traducción a código máquina del programa. Debes tener presente que dicho fichero sólo se genera si el programa C está correctamente escrito.

Si queremos compilar el programa `sumatorio.c` hemos de usar una opción especial:

```
gcc sumatorio.c -lm -o sumatorio
```

La opción `-lm` se debe usar siempre que nuestro programa utilice funciones del módulo matemático (como *sqrt*, que se usa en `sumatorio.c`). Ya te indicaremos por qué en la sección dedicada a presentar el módulo matemático de C.

C99 y gcc

Por defecto, `gcc` acepta programas escritos en C89 con extensiones introducidas por GNU (el grupo de desarrolladores de muchas herramientas de Linux). Muchas de esas extensiones de GNU forman ya parte de C99, así que `gcc` es, por defecto, el compilador de un lenguaje intermedio entre C89 y C99. Si en algún momento da un aviso indicando que no puede compilar algún programa porque usa características propias del C99 no disponibles por defecto, puedes forzarle a compilar en «modo C99» así:

```
gcc programa.c -std=c99 -o programa
```

Has de saber, no obstante, que `gcc` aún no soporta el 100% de C99 (aunque sí todo lo que te explicamos en este texto).

El compilador `gcc` acepta muchas otras variantes de C. Puedes forzarle a aceptar una en particular «asignando» a la opción `-std` el valor `c89`, `c99`, `gnu89` o `gnu99`.

1.2. Traduciendo de Python a C: una guía rápida

Empezaremos por presentar de forma concisa cómo traducir la mayor parte de los programas Python que aprendimos a escribir en los capítulos 3 y 4 del primer volumen a programas equivalentes en C. En secciones posteriores entraremos en detalle y nos dedicaremos a estudiar las muchas posibilidades que ofrece C a la hora de seleccionar tipos de datos, presentar información con sentencias de impresión en pantalla, etc.

[2] La versión 3.2 de `gcc` es la primera en ofrecer un soporte suficiente de C99. Si usas una versión anterior, es posible que algunos (pocos) programas del libro no se compilen correctamente.

1. Los programas (sencillos) presentan, generalmente, este aspecto:

```
#include <stdio.h>

Posiblemente otros «#include»

int main(void)
{
  Programa principal.

  return 0;
}
```

   Hay, pues, dos zonas: una inicial cuyas líneas empiezan por **#include** (equivalentes a las sentencias **import** de Python) y una segunda que empieza con una línea «**int** *main*(**void**)» y comprende las sentencias del programa principal mas una línea «**return** 0;», encerradas todas ellas entre llaves ({ y }).

   De ahora en adelante, todo texto comprendido entre llaves recibirá el nombre de *bloque*.

2. Toda variable debe declararse antes de ser usada. La declaración de la variable consiste en escribir el nombre de su tipo (**int** para enteros y **float** para flotantes)[3] seguida del identificador de la variable y un punto y coma. Por ejemplo, si vamos a usar una variable entera con identificador *a* y una variable flotante con identificador *b*, nuestro programa las declarará así:

```
#include <stdio.h>

int main(void)
{
  int a;
  float b;

  Sentencias donde se usan las variables.

  return 0;
}
```

   No es obligatorio que la declaración de las variables tenga lugar justo al principio del bloque que hay debajo de la línea «**int** *main*(**void**)», pero sí conveniente.[4]

   Si tenemos que declarar dos o más variables del mismo tipo, podemos hacerlo en una misma línea separando los identificadores con comas. Por ejemplo, si las variables *x*, *y* y *z* son todas de tipo **float**, podemos recurrir a esta forma compacta de declaración:

```
#include <stdio.h>

int main(void)
{
  float x, y, z;

  ...
```

[3]Recuerda que no estudiaremos las variables de tipo cadena hasta el próximo capítulo.

[4]En versiones de C anteriores a C99 sí era obligatorio que las declaraciones se hicieran al principio de un bloque. C99 permite declarar una variable en cualquier punto del programa, siempre que éste sea anterior al primer uso de la misma.

```

    return 0;
}
```

3. Las sentencias de asignación C son similares a las sentencias de asignación Python: a mano izquierda del símbolo igual (=) se indica la variable a la que se va a asignar el valor que resulta de evaluar la expresión que hay a mano derecha. Cada sentencia de asignación debe finalizar con punto y coma.

```c
#include <stdio.h>

int main(void)
{
    int a;
    float b;

    a = 2;
    b = 0.2;

    return 0;
}
```

Como puedes ver, los números enteros y flotantes se representan igual que en Python.

4. Las expresiones se forman con los mismos operadores que aprendimos en Python. Bueno, hay un par de diferencias:

   - Los operadores Python **and**, **or** y **not** se escriben en C, respectivamente, con &&, || y !;
   - No hay operador de exponenciación (que en Python era **).
   - Hay operadores para la conversión de tipos. Si en Python escribíamos *float*(*x*) para convertir el valor de *x* a flotante, en C escribiremos (**float**) *x* para expresar lo mismo. Fíjate en cómo se disponen los paréntesis: los operadores de conversión de tipos son de la forma (**tipo**).

```c
#include <stdio.h>

int main(void)
{
    int a;
    float b;

    a = 13 % 2;
    b = 2.0 / (1.0 + 2 - (a + 1));

    return 0;
}
```

Las reglas de asociatividad y precedencia de los operadores son casi las mismas que aprendimos en Python. Hay más operadores en C y los estudiaremos más adelante.

5. Para mostrar resultados por pantalla se usa la función *printf*. La función recibe uno o más argumentos separados por comas:

- primero, una cadena con formato, es decir, con marcas de la forma %d para representar enteros y marcas %f para representar flotantes (en los que podemos usar modificadores para, por ejemplo, controlar la cantidad de espacios que ocupará el valor o la cantidad de cifras decimales de un número flotante);
- y, a continuación, las expresiones cuyos valores se desea mostrar (debe haber una expresión por cada marca de formato).

escribe.c

```c
#include <stdio.h>

int main(void)
{
  int a;
  float b;

  a = 13 % 2;
  b = 2.0 / (1.0 + 2 - (a + 1));

  printf("El valor de a es %d y el de b es %f\n", a, b);

  return 0;
}
```

La cadena con formato debe ir encerrada entre comillas dobles, no simples. El carácter de retorno de carro (\n) es obligatorio si se desea finalizar la impresión con un salto de línea. (Observa que, a diferencia de Python, no hay operador de formato entre la cadena de formato y las expresiones: la cadena de formato se separa de la primera expresión con una simple coma).

Como puedes ver, todas las sentencias de los programas C que estamos presentando finalizan con punto y coma.

6. Para leer datos de teclado has de usar la función *scanf*. Fíjate en este ejemplo:

lee_y_escribe.c

```c
1  #include <stdio.h>
2  
3  int main(void)
4  {
5    int a;
6    float b;
7  
8    scanf("%d", &a);
9    scanf("%f", &b);
10 
11   printf("El valor de a es %d y el de b es %f\n", a, b);
12 
13   return 0;
14 }
```

La línea 8 lee de teclado el valor de un entero y lo almacena en *a*. La línea 9 lee de teclado el valor de un flotante y lo almacena en *b*. Observa el uso de marcas de formato en el primer argumento de *scanf*: %d señala la lectura de un **int** y %f la de un **float**. El símbolo & que precede al identificador de la variable en la que se almacena el valor leído es obligatorio para variables de tipo escalar.

Si deseas mostrar por pantalla un texto que proporcione información acerca de lo que el usuario debe introducir, hemos de usar nuevas sentencias *printf*:

lee_mejor_y_escribe.c   lee_mejor_y_escribe.c

```c
#include <stdio.h>

int main(void)
{
   int a;
   float b;

   printf("Introduce un entero a: ");
   scanf("%d", &a);
   printf("Y ahora un flotante b: ");
   scanf("%f", &b);

   printf("El valor de a es %d y el de b es %f\n", a, b);

   return 0;
}
```

7. La sentencia **if** de Python presenta un aspecto similar en C:

si_es_par.c   si_es_par.c

```c
#include <stdio.h>

int main(void)
{
   int a;

   printf("Introduce un entero a: ");
   scanf("%d", &a);

   if (a % 2 == 0) {
      printf("El valor de a es par.\n");
      printf("Es curioso.\n");
   }

   return 0;
}
```

Ten en cuenta que:

- la condición va encerrada obligatoriamente entre paréntesis;
- y el bloque de sentencias cuya ejecución está supeditada a la satisfacción de la condición va encerrado entre llaves (aunque matizaremos esta afirmación más adelante).

Naturalmente, puedes anidar sentencias **if**.

si_es_par_y_positivo.c   si_es_par_y_positivo.c

```c
#include <stdio.h>

int main(void)
{
   int a;

   printf("Introduce un entero a: ");
   scanf("%d", &a);

   if (a % 2 == 0) {
```

```c
        printf("El valor de a es par.\n");
        if (a > 0) {
            printf("Y, además, es positivo.\n");
        }
    }

    return 0;
}
```

También hay sentencia **if-else** en C:

par_o_impar.c **par_o_impar.c**

```c
#include <stdio.h>

int main(void)
{
    int a;

    printf("Introduce un entero a: ");
    scanf("%d", &a);

    if (a % 2 == 0) {
        printf("El valor de a es par.\n");
    }
    else {
        printf("El valor de a es impar.\n");
    }

    return 0;
}
```

No hay, sin embargo, sentencia **if-elif**, aunque es fácil obtener el mismo efecto con una sucesión de **if-else if**:

tres_casos.c **tres_casos.c**

```c
#include <stdio.h>

int main(void)
{
    int a;

    printf("Introduce un entero a: ");
    scanf("%d", &a);

    if (a > 0) {
        printf("El valor de a es positivo.\n");
    }
    else if (a == 0) {
        printf("El valor de a es nulo.\n");
    }
    else if (a < 0) {
        printf("El valor de a es negativo.\n");
    }
    else {
        printf("Es imposible mostrar este mensaje.\n");
    }

    return 0;
}
```

8. La sentencia **while** de C es similar a la de Python, pero has de tener en cuenta la obligatoriedad de los paréntesis alrededor de la condición y que las sentencias que se pueden repetir van encerradas entre un par de llaves:

cuenta_atras.c cuenta_atras.c

```
#include <stdio.h>

int main(void)
{
  int a;

  printf("Introduce␣un␣entero␣a:␣");
  scanf("%d", &a);

  while (a > 0) {
    printf("%d", a);
    a -= 1;
  }
  printf("¡Boom!\n");

  return 0;
}
```

9. También puedes usar la sentencia **break** en C:

primo.c primo.c

```
#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Introduce␣un␣entero␣a:␣");
  scanf("%d", &a);

  b = 2;
  while (b < a) {
    if (a % b == 0) {
      break;
    }
    b += 1;
  }
  if (b == a) {
    printf("%d␣es␣primo.\n", a);
  }
  else {
    printf("%d␣no␣es␣primo.\n", a);
  }

  return 0;
}
```

10. Los módulos C reciben el nombre de *bibliotecas* y se importan con la sentencia **#include**. Ya hemos usado **#include** en la primera línea de todos nuestros programas: **#include** `<stdio.h>`. Gracias a ella hemos importado las funciones de entrada/salida *scanf* y *printf*. No se puede importar una sola función de una biblioteca: debes importar el contenido completo de la biblioteca.

Las funciones matemáticas pueden importarse del módulo matemático con **#include** `<math.h>` y sus nombres son los mismos que vimos en Python (*sin* para el seno, *cos* para el coseno, etc.).

raiz_cuadrada.c

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  float b;

  printf("Escribe un flotante: ");
  scanf("%f", &b);

  if (b >= 0.0) {
    printf("Su raíz cuadrada es %f.\n", sqrt(b));
  }
  else {
    printf("No puedo calcular su raíz cuadrada.\n");
  }

  return 0;
}
```

No hay funciones predefinidas en C. Muchas de las que estaban predefinidas en Python pueden usarse en C, pero importándolas de bibliotecas. Por ejemplo, *abs* (valor absoluto) puede importarse del módulo `stdlib.h` (por «standard library», es decir, «biblioteca estándar»).

Las (aproximaciones a las) constantes π y e se pueden importar de la biblioteca matemática, pero sus identificadores son ahora `M_PI` y `M_E`, respectivamente.

No está mal: ya sabes traducir programas Python sencillos a C (aunque no sabemos traducir programas con definiciones de función, ni con variables de tipo cadena, ni con listas, ni con registros, ni con acceso a ficheros...). ¿Qué tal practicar con unos pocos ejercicios?

EJERCICIOS

▶ 2 Traduce a C este programa Python.

```
a = int(raw_input('Dame el primer número: '))
b = int(raw_input('Dame el segundo número: '))

if a >= b:
  maximo = a
else:
  maximo = b

print 'El máximo es', maximo
```

▶ 3 Traduce a C este programa Python.

```
n = int(raw_input('Dame un número: '))
m = int(raw_input('Dame otro número: '))

if n * m == 100:
  print 'El producto %d * %d es igual a 100' % (n, m)
else:
  print 'El producto %d * %d es distinto de 100' % (n, m)
```

▶ 4 Traduce a C este programa Python.

```
from math import sqrt

x1 = float(raw_input("Punto 1, coordenada x: "))
y1 = float(raw_input("Punto 1, coordenada y: "))
x2 = float(raw_input("Punto 2, coordenada x: "))
y2 = float(raw_input("Punto 2, coordenada y: "))
dx = x2 - x1
dy = y2 - y1
distancia = sqrt(dx**2 + dy**2)
print 'La distancia entre los puntos es: ', distancia
```

▶ 5 Traduce a C este programa Python.

```
a = float(raw_input('Valor de a: '))
b = float(raw_input('Valor de b: '))

if a != 0:
  x = -b/a
  print 'Solución: ', x
else:
  if b != 0:
    print 'La ecuación no tiene solución.'
  else:
    print 'La ecuación tiene infinitas soluciones.'
```

▶ 6 Traduce a C este programa Python.

```
from math import log

x = 1.0
while x < 10.0:
  print x, '\t', log(x)
  x = x + 1.0
```

▶ 7 Traduce a C este programa Python.

```
n = 1
while n < 6:
  i = 1
  while i < 6:
    print n*i, '\t',
    i = i + 1
  print
  n = n + 1
```

▶ 8 Traduce a C este programa Python.

```
from math import pi

opcion = 0
while opcion != 4:
  print 'Escoge una opción: '
  print '1) Calcular el diámetro.'
  print '2) Calcular el perímetro.'
  print '3) Calcular el área.'
  print '4) Salir.'
  opcion = int(raw_input('Teclea 1, 2, 3 o 4 y pulsa el retorno de carro: '))

  radio = float(raw_input('Dame el radio de un círculo: '))

```

```
  if opcion == 1:
    diametro = 2 * radio
    print 'El␣diámetro␣es', diametro
  elif opcion == 2:
    perimetro = 2 * pi * radio
    print 'El␣perímetro␣es', perimetro
  elif opcion == 3:
    area = pi * radio ** 2
    print 'El␣área␣es', area
  elif opcion < 0 or opcion > 4:
    print 'Sólo␣hay␣cuatro␣opciones:␣1,␣2,␣3␣o␣4.␣Tú␣has␣tecleado', opcion
```

Ya es hora, pues, de empezar con los detalles de C.

1.3. Estructura típica de un programa C

Un programa C no es más que una colección de declaraciones de variables globales y de definiciones de constantes, macros, tipos y funciones. Una de las funciones es especial: se llama *main* (que en inglés significa «principal») y contiene el código del programa principal. No nos detendremos a explicar la sintaxis de la definición de funciones hasta el capítulo 3, pero debes saber ya que la definición de la función *main* empieza con «**int** *main* (**void**)» y sigue con el cuerpo de la función encerrado entre un par de llaves. La función *main* debe devolver un valor entero al final (típicamente el valor 0), por lo que finaliza con una sentencia **return** que devuelve el valor 0.[5]

La estructura típica de un programa C es ésta:

```
Importación de funciones, variables, constantes, etc.

Definición de constantes y macros.

Definición de nuevos tipos de datos.

Declaración de variables globales.

Definición de funciones.

int main(void)
{
  Declaración de variables propias del programa principal (o sea, locales a main).

  Programa principal.

  return 0;
}
```

Un fichero con extensión «.c» que no define la función main no es un programa C completo.

Si, por ejemplo, tratamos de compilar este programa incorrecto (no define *main*):

```
sin_main.c
int a;
a = 1;
```

[5] El valor 0 se toma, por un convenio, como señal de que el programa finalizó correctamente. El sistema operativo Unix recibe el valor devuelto con el **return** y el intérprete de órdenes, por ejemplo, puede tomar una decisión acerca de qué hacer a continuación en función del valor devuelto.

el compilador muestra el siguiente mensaje (u otro similar, según la versión del compilador que utilices):

```
$ gcc sin_main.c -o sin_main ↲
sin_main.c:2: warning: data definition has no type or storage class
/usr/lib/crt1.o: En la función ‘_start’:
/usr/lib/crt1.o(.text+0x18): referencia a ‘main’ sin definir
collect2: ld returned 1 exit status
```

Fíjate en la tercera línea del mensaje de error: «`referencia a ‘main’ sin definir`».

1.4. C es un lenguaje de formato libre

Así como en Python la indentación determina los diferentes bloques de un programa, en C la indentación es absolutamente superflua: indentamos los programas únicamente para hacerlos más legibles. En C se sabe dónde empieza y dónde acaba un bloque porque éste está encerrado entre una llave abierta ({) y otra cerrada (}).

He aquí un ejemplo de bloques anidados en el que hemos indentado el código para facilitar su lectura:

minimo.c — minimo.c

```c
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b) {
    if (a < c) {
      minimo = a;
    }
    else {
      minimo = c;
    }
  }
  else {
    if (b < c) {
      minimo = b;
    }
    else {
      minimo = c;
    }
  }
  printf("%d\n", minimo);
  return 0;
}
```

Este programa podría haberse escrito como sigue y sería igualmente correcto:

minimo_1.c — minimo.c

```c
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;
```

```

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b) {
    if (a < c) { minimo = a; }
    else { minimo = c; }
  }
  else {
    if (b < c) { minimo = b; }
    else { minimo = c; }
  }
  printf("%d\n", minimo);
  return 0;
}
```

Cuando un bloque consta de una sola sentencia no es necesario encerrarla entre llaves. Aquí tienes un ejemplo:

minimo_2.c — minimo.c

```
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b) {
    if (a < c) minimo = a;
    else minimo = c;
  }
  else {
    if (b < c) minimo = b;
    else minimo = c;
  }
  printf("%d\n", minimo);
  return 0;
}
```

De hecho, como **if-else** es una única sentencia, también podemos suprimir las llaves restantes:

minimo_3.c — minimo.c

```
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b)
    if (a < c) minimo = a;
    else minimo = c;
  else
    if (b < c) minimo = b;
    else minimo = c;
```

```c
  printf("%d\n", minimo);
  return 0;
}
```

Debes tener cuidado, no obstante, con las ambigüedades que parece producir un sólo **else** y dos **if**:

primero_es_minimo_1.c — primero_es_minimo.c

```c
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b)
    if (a < c)
      printf("El primero es el mínimo.\n");
  else
    printf("El primero no menor que el segundo.\n");
  printf("%d\n", minimo);
  return 0;
}
```

¿Cuál de los dos **if** se asocia al **else**? C usa una regla: el **else** se asocia al **if** más próximo (en el ejemplo, el segundo). Según esa regla, el programa anterior no es correcto. El sangrado sugiere una asociación entre el primer **if** y el **else** que no es la que interpreta C. Para que C «entienda» la intención del autor es necesario que explicites con llaves el alcance del primer **if**:

primero_es_minimo_2.c — primero_es_minimo.c

```c
#include <stdio.h>

int main(void)
{
  int a, b, c, minimo;

  scanf("%d", &a);
  scanf("%d", &b);
  scanf("%d", &c);
  if (a < b) {
    if (a < c)
      printf("El primero es el mínimo.\n");
  }
  else
    printf("El primero no es menor que el segundo.\n");
  printf("%d\n", minimo);
  return 0;
}
```

Ahora que has adquirido la práctica de indentar los programas gracias a la disciplina impuesta por Python, síguela siempre, aunque programes en C y no sea necesario.

Una norma: *las sentencias C acaban con un punto y coma*. Y una excepción a la norma: no hace falta poner punto y coma tras una llave cerrada.[6]

Dado que las sentencias finalizan con punto y coma, no tienen por qué ocupar una línea. Una sentencia como «$a = 1$;» podría escribirse, por ejemplo, en cuatro líneas:

---

[6]Habrá una excepción a esta norma: las construcciones **struct**, cuya llave de cierre debe ir seguida de un punto y coma.

La indentación no importa... pero nadie se pone de acuerdo

En C no es obligatorio indentar los programas, aunque todos los programadores están de acuerdo en que un programa sin una «correcta» indentación es ilegible. ¡Pero no hay consenso en lo que significa indentar «correctamente»! Hay varios estilos de indentación en C y cada grupo de desarrolladores escoge el que más le gusta. Te presentamos unos pocos estilos:

a) La llave abierta se pone en la misma línea con la estructura de control y la llave de cierre va en una línea a la altura del inicio de la estructura:

```
if (a==1) {
   b = 1;
   c = 2;
}
```

b) Ídem, pero la llave de cierre se dispone un poco a la derecha:

```
if (a==1) {
   b = 1;
   c = 2;
   }
```

c) La llave abierta va en una línea sola, al igual que la llave cerrada. Ambas se disponen a la altura de la estructura que gobierna el bloque:

```
if (a==1)
{
   b = 1;
   c = 2;
}
```

d) Ídem, pero las dos llaves se disponen más a la derecha y el contenido del bloque más a la derecha:

```
if (a==1)
   {
      b = 1;
      c = 2;
   }
```

e) Y aún otro, con las llaves a la misma altura que el contenido del bloque:

```
if (a==1)
   {
   b = 1;
   c = 2;
   }
```

No hay un estilo mejor que otro. Es cuestión de puro convenio. Aún así, hay más de una discusión subida de tono en los grupos de debate para desarrolladores de C. Increíble, ¿no? En este texto hemos optado por el primer estilo de la lista (que, naturalmente, es el «correcto» ;-)) para todas las construcciones del lenguaje a excepción de la definición de funciones (como *main*), que sigue el convenio de indentación que relacionamos en tercer lugar.

```
a
=
1
;
```

Pero aunque sea lícito escribir así esa sentencia, no tienen ningún sentido y hace más difícil la comprensión del programa. Recuerda: vela siempre por la legibilidad de los programas.

También podemos poner más de una sentencia en una misma línea, pues el compilador sabrá dónde empieza y acaba cada una gracias a los puntos y comas, las llaves, etc. El programa `sumatorio.c`, por ejemplo, podría haberse escrito así:

sumatorio_ilegible.c — **sumatorio_ilegible.c**

```
#include <stdio.h>
#include <math.h>
int main(void) { int a, b, i; float s; /* Pedir límites inferior y superior. */ printf(
"Límite␣inferior:"); scanf("%d", &a); while (a < 0) { printf(
"No␣puede␣ser␣negativo\n"); printf("Límite␣inferior:"); scanf("%d", &a);}
printf("Límite␣superior:"); scanf("%d", &b); while (b < a) { printf(
"No␣puede␣ser␣mayor␣que␣%d\n", a); printf("Límite␣superior:"); scanf("%d",
&b); } /* Calcular el sumatorio de la raíz cuadrada de i para i entre
a y b. */ s = 0.0; for (i = a; i <= b; i++) { s += sqrt(i); } /* Mostrar
el resultado. */ printf("Sumatorio␣de␣raíces␣"); printf("de␣%d␣a␣%d:␣%f\n",
a, b, s); return 0;}
```

Obviamente, hubiera sido una mala elección: un programa escrito así, aunque correcto, es completamente ilegible.[7]

Un programador de C experimentado hubiera escrito *sumatorio.c* utilizando llaves sólo donde resultan necesarias y, probablemente, utilizando unas pocas líneas menos. Estudia las diferencias entre la primera versión de *sumatorio.c* y esta otra:

sumatorio_1.c — **sumatorio.c**

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  int a, b, i;
  float s;

  /* Pedir límites inferior y superior. */
  printf("Límite␣inferior:"); scanf("%d", &a);
  while (a < 0) {
    printf("No␣puede␣ser␣negativo\nLímite␣inferior:"); scanf("%d", &a);
  }

  printf("Límite␣superior:"); scanf("%d", &b);
  while (b < a) {
    printf("No␣puede␣ser␣mayor␣que␣%d\nLímite␣superior:", a); scanf("%d",&b);
  }

  /* Calcular el sumatorio de la raíz cuadrada de i para i entre a y b. */
  s = 0.0;
  for (i = a; i <= b; i++) s += sqrt(i);

  /* Mostrar el resultado. */
  printf("Sumatorio␣de␣raíces␣de␣%d␣a␣%d:␣%f\n", a, b, s);

  return 0;
}
```

[7]Quizá hayas reparado en que las líneas que empiezan con **#include** son especiales y que las tratamos de forma diferente: no se puede jugar con su formato del mismo modo que con las demás: cada sentencia **#include** debe ocupar una línea y el carácter # debe ser el primero de la línea.

International Obfuscated C Code Contest

Es posible escribir programas ilegibles en C, ¡hasta tal punto que hay un concurso internacional de programas ilegibles!: el International Obfuscated C Code Contest (IOCCC). Este programa (en K&R C, ligeramente modificado para que compile con `gcc`) concursó en 1989:

```
                                                   extern int
                                                       errno
                                                         ;char
                                                            grrr
                             ;main(                           r,
  argv, argc )            int    argc                           ,
   r        ;           char *argv[];{int                     P( );
#define x int i=0,     j=0,cc[4];printf("      choo choo\n"     ) ;
x  ;if    (P(  !        i              )        |  cc[  !      j ]
&  P(j    )>2  ?        j              :        i  ){*  argv[i++ +!-i]
;              for    (i=              0;;    i++                   );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

¿Sabes qué hace? ¡Sólo imprime en pantalla «`choo choo`»! Este otro, de la edición de 1992, es un generador de anagramas escrito por Andreas Gustafsson (AG `;-)`):

```
#include <stdio.h>

long a
[4],b[
4],c[4]
,d[0400],e=1;
typedef struct f{long g
,h,i[4]    ,j;struct f*k;}f;f g,*
l[4096               ]; char h[256],*m,k=3;
                     long n     (o, p,q)long*o,*p,*q;{
                     long r               =4,s,i=0;for(;r--;s=i^
                     *o^*p,                         i=i&*p|(i|*p)&~*o++,*q
                     ++=s,p                         ++);return i;}t(i,p)long*p
                     ;{*c=d               [i],n(a,c,b),n(p,b,p);}u(j)f*j;{j->h
                     =(j->g     =j->i[0]|j->i[1]|j->i[2]|j->i[3])&4095;}v(
j,s)f*               j; {int i; for(j->k->k&&v(j->k, ' '),fseek(
stdin,     j->j, 0);i=getchar(),putchar(i-'\n'?i:s),i-
'\n';);}w(o,r,j,x,p)f*o,*j;long p;{f q;int
s,i=o->h;q.k=o;r>i?j=l[r=i]:r<i&&
(s=r&~i)?(s|=s>>1, s|=s
>>2,s|=s>>4,s
|=s>>8
,j=l[r
=((r&i                   |s)&~(s>>1))-1&i]):0;--x;for
                 (;x&&!(p&i);p>>=1);for(;!x&&j;n(o->i,j->i,q.
            i),u(&q),q.g||(q.j=j->j,v(&q,'\n')),j=j->k);for(;x;j=x
         ?j->k:0){for(;!j&&((r=(r&i)-1&i)-i&&(r&p)?2:(x=0));j=l[r]);!
      x||(j->g&~o->g)||n                              (o->i,j->i,q.i)||(
    u(&q), q.j=j                                              ->j,q.g?w(&q
   ,r,j->k,x                                                      ,p):v(&q,
  '\n'));                                                            }}y(){f
 j;char                                                                *z,*p;
for(;m                                                                  ? j.j=
ftell(                                                                  stdin)
,7,(m=                       gets(m                                     ))||w(
&g,315                       *13,l[                                     4095]
 ,k,64*                      64)&0:                                    0;n(g
  .i,j.i,                    b)||(u                                 (&j),j.
   k=l[j.h],l[j.h]= &j,y())){for(z=                             p=h;*z&&(
    d[*z++]||(p=0));for(z=p?n(j.i                       ,j.i,j.i)+h:"";
  *z;t(*z++,j.i));}}main(o,p)char**              p; {for(;m = *++p;)for(;*m-
'-'?*m:(k= -atoi(m))&0;d[*m]||(d[*m              ]=e,e<<=1),t(*m++,g.i)); u(&
                             g),m=h
                             ,y();}
```

El programa lee un diccionario de la entrada estándar y recibe el número de palabras del anagrama (precedido por un guión) y el texto del que se desea obtener un anagrama. Si compilas el programa y lo ejecutas así descubrirás algunos anagramas curiosos

```
$ anagrama </usr/dict/words -3 universitat jaume i) ↲
```

Por pantalla aparecerán decenas de anagramas, entre ellos «`autism injure vitae`» y «`mutate via injuries`». Usando un diccionario español y diferentes números de palabras obtendrás, entre otros, éstos: «`mutis, vieja uterina`» o «`mi jeta nueva, tu iris`».

Ya sabes: *puedes* escribir programas ilegibles en C. ¡Procura que tus programas no merezcan una mención de honor en el concurso!

Los lenguajes de programación en los que el código no debe seguir un formato determinado de líneas y/o bloques se denominan *de formato libre*. Python no es un lenguaje de formato libre; C sí.

. . . . EJERCICIOS . . . .

▶ **9** Este programa C incorrecto tiene varios errores que ya puedes detectar. Indica cuáles son:

```
#include <stdio.h>

int a, b;

scanf("%d", &a); scanf("%d", &b)
while (a <= b):
  scanf("%d", &a)
  scanf("%d", &b)
printf("%d %d\n", a, b);
```

▶ **10** Indenta «correctamente» este programa C.

```
#include <stdio.h>
int main(void)
{
int a, b;
scanf("%d", &a);
scanf("%d", &b);
while(a > b) {
scanf("%d", &a);
scanf("%d", &b);
}
printf("%d %d\n", a, b);
return 0;
}
```

. . . .

1.5. Hay dos tipos de comentario

C99 permite escribir comentarios de dos formas distintas. Una es similar a la de Python: se marca el inicio de comentario con un símbolo especial y éste se prolonga hasta el final de línea. La marca especial no es #, sino //. El segundo tipo de comentario puede ocupar más de una línea: empieza con los caracteres /* y finaliza con la primera aparición del par de caracteres */.

En este ejemplo aparecen comentarios que abarcan más de una línea:

maximo.c

```
/*********************************************************************
 * Un programa de ejemplo.
 *-----------------------------------
 * Propósito: mostrar algunos efectos que se pueden lograr con
 * comentarios de C
 *********************************************************************/
#include <stdio.h>

/*-----------------------------------
 * Programa principal
 *-----------------------------------*/

```

```
15 int main(void)
16 {
17   int a, b, c; // Los tres números.
18   int m;       // Variable para el máximo de los tres.
19
20   /* Lectura de un número */
21   printf("a:␣"); scanf("%d", &a);
22   /* ... de otro ... */
23   printf("b:␣"); scanf("%d", &b);
24   /* ... y de otro más. */
25   printf("c:␣"); scanf("%d", &c);
26   if (a > b)
27     if (a > c) //En este caso a > b y a > c.
28       m = a;
29     else //Y en este otro caso b < a ≤ c.
30       m = c;
31   else
32     if (b > c) //En este caso a ≤ b y b > c.
33       m = b;
34     else //Y en este otro caso a ≤ b ≤ c.
35       m = c;
36   /* Impresión del resultado. */»
37   printf("El␣máximo␣de␣%d,␣%d␣y␣%d␣es␣%d\n", a, b, c, m);
38   return 0;
39 }
```

Uno de los comentarios empieza al principio de la línea 1 y finaliza al final de la líne 6 (sus dos últimos caracteres visibles son un asterisco y una barra). Hay otro que empieza en la línea 10 y finaliza en al línea 12. Y hay otros que usan las marcas `/*` y `*/` en líneas como la 20 o la 22, aunque hubiésemos podidos usar en ambos casos la marca `//`.

Los comentarios encerrados entre `/*` y `*/` no se pueden anidar. Este fragmento de programa es incorrecto:

```
/* un /* comentario */ mal hecho */
```

¿Por qué? Parece que hay un comentario dentro de otro, pero no es así: el comentario que empieza en el primer par de caracteres `/*` acaba en el primer par de caracteres `*/`, no en el segundo. El texto del *único* comentario aparece aquí enmarcado:

```
/* un /* comentario */ mal hecho */
```

Así pues, el fragmento « `mal hecho */`» no forma parte de comentario alguno y no tiene sentido en C, por lo que el compilador detecta un error.

. . . . EJERCICIOS . . . .

▶ 11 Haciendo pruebas durante el desarrollo de un programa hemos decidido comentar una línea del programa para que, de momento, no sea compilada. El programa nos queda así:

```
#include <stdio.h>

int main(void)
{
  int a, b, i, j;

  scanf("%d", &a);
  scanf("%d", &b);
  i = a;
```

```
  j = 1;
  while (i <= b) {
    /* printf("%d %d\n", i, j); */
    j *= 2;
    i += 1;
  }
  printf("%d\n", j);
  return 0;
}
```

Compilamos el programa y el compilador no detecta error alguno. Ahora decidimos comentar el bucle **while** completo, así que añadimos un nuevo par de marcas de comentario (líneas 11 y 17):

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b, i, j;
6
7    scanf("%d", &a);
8    scanf("%d", &b);
9    i = a;
10   j = 1;
11   /*
12    while (i <= b) {
13      /* printf("%d %d\n", i, j); */
14      j *= 2;
15      i += 1;
16    }
17    */
19   printf("%d\n", j);
20   return 0;
21 }
```

Al compilar nuevamente el programa aparecen mensajes de error. ¿Por qué?

▶ 12 ¿Da problemas este otro programa con comentarios?

```
#include <stdio.h>

int main(void)
{
  int a, b, i, j;

  scanf("%d", &a);
  scanf("%d", &b);
  i = a;
  j = 1;
  /*
   while (i <= b) {
     // printf("%d %d\n", i, j);
     j *= 2;
     i += 1;
   }
   */
  printf("%d\n", j);
  return 0;
}
```

▶ 13 ¿Cómo se interpreta esta sentencia?

```
i = x //*y*/z++
;
```

1.6. Valores literales en C

Por *valores literales* nos referimos a valores de números, cadenas y caracteres dados explícitamente. Afortunadamente, las reglas de escritura de literales en C son similares a las de Python.

1.6.1. Enteros

Una forma natural de expresar un número entero en C es mediante una secuencia de dígitos. Por ejemplo, 45, 0 o 124653 son enteros. Al igual que en Python, está prohibido insertar espacios en blanco (o cualquier otro símbolo) entre los dígitos de un literal entero.

Hay más formas de expresar enteros. En ciertas aplicaciones resulta útil expresar un número entero en base 8 (sistema octal) o en base 16 (sistema hexadecimal). Si una secuencia de dígitos empieza en 0, se entiende que codifica un número en base 8. Por ejemplo, 010 es el entero 8 (en base 10) y 0277 es el entero 191 (en base 10). Para codificar un número en base 16 debes usar el par de caracteres 0x seguido del número en cuestión. El literal 0xff, por ejemplo, codifica el valor decimal 255.

Pero aún hay una forma más de codificar un entero, una que puede resultar extraña al principio: mediante un carácter entre comillas simples, que representa a su valor ASCII. El valor ASCII de la letra «a minúscula», por ejemplo, es 97, así que el literal `'a'` es el valor 97. Hasta tal punto es así que podemos escribir expresiones como `'a'`+1, que es el valor 98 o, lo que es lo mismo, `'b'`.

Se puede utilizar cualquiera de las secuencias de escape que podemos usar con las cadenas. El literal `'\n'`, por ejemplo, es el valor 10 (que es el código ASCII del salto de línea).

> **Ni ord ni chr**
>
> En C no son necesarias las funciones *ord* o *chr* de Python, que convertían caracteres en enteros y enteros en caracteres. Como en C los caracteres son enteros, no resulta necesario efectuar conversión alguna.

1.6.2. Flotantes

Los números en coma flotante siguen la misma sintaxis que los flotantes de Python. Un número flotante debe presentar parte decimal y/o exponente. Por ejemplo, 20.0 es un flotante porque tiene parte decimal (aunque sea nula) y 2e1 también lo es, pero porque tiene exponente (es decir, tiene una letra *e* seguida de un entero). Ambos representan al número real 20.0. (Recuerda que 2e1 es $2 \cdot 10^1$.) Es posible combinar en un número parte decimal con exponente: 2.0e1 es un número en coma flotante válido.

1.6.3. Cadenas

Así como en Python puedes optar por encerrar una cadena entre comillas simples o dobles, en C sólo puedes encerrarla entre comillas dobles. Dentro de las cadenas puedes

utilizar secuencias de escape para representar caracteres especiales. Afortunadamente, las secuencias de escape son las mismas que estudiamos en Python. Por ejemplo, el salto de línea es `\n` y la comilla doble es `\"`.

. . . . EJERCICIOS . . . .

▶ **14** Traduce a cadenas C las siguientes cadenas Python:

1. `"una␣cadena"`
2. `'una␣cadena'`
3. `"una␣\"cadena\""`
4. `'una␣"cadena"'`
5. `'una␣\'cadena\''`
6. `"una␣cadena␣que␣ocupa\n␣dos␣líneas"`
7. `"una␣cadena␣que␣\\no␣ocupa␣dos␣líneas"`

. . . .

Te relacionamos las secuencias de escape que puedes necesitar más frecuentemente:

Secuencia	Valor
`\a`	(alerta): produce un aviso audible o visible.
`\b`	(*backspace*, espacio atrás): el cursor retrocede un espacio a la izquierda.
`\f`	(*form feed*, alimentación de página): pasa a una nueva «página».
`\n`	(*newline*, nueva línea): el cursor pasa a la primera posición de la siguiente línea.
`\r`	(*carriage return*, retorno de carro): el cursor pasa a la primera posición de la línea actual.
`\t`	(tabulador): desplaza el cursor a la siguiente marca de tabulación.
`\\`	muestra la barra invertida.
`\"`	muestra la comilla doble.
`\`*número_octal*	muestra el carácter cuyo código ASCII (o IsoLatin) es el número octal indicado. El número octal puede tener uno, dos o tres dígitos octales. Por ejemplo `"\60"` equivale a `"0"`, pues el valor ASCII del carácter cero es 48, que en octal es 60.
`\x`*número_hexadecimal*	ídem, pero el número está codificado en base 16 y puede tener uno o dos dígitos hexadecimales. Por ejemplo, `"\x30"` también equivale a `"0"`, pues 48 en decimal es 30 en hexadecimal.
`\?`	muestra el interrogante.

Es pronto para aprender a utilizar variables de tipo cadena. Postergamos este asunto hasta el apartado 2.2.

1.7. C tiene un rico juego de tipos escalares

En Python tenemos dos tipos numéricos escalares: enteros y flotantes[8]. En C hay una gran variedad de tipos escalares en función del número de cifras o de la precisión con

[8]Bueno, esos son los que hemos estudiado. Python tiene, además, enteros largos. Otro tipo numérico no secuencial de Python es el complejo.

la que deseamos trabajar, así que nos permite tomar decisiones acerca del compromiso entre rango/precisión y ocupación de memoria: a menor rango/precisión, menor ocupación de memoria.

No obstante, nosotros limitaremos nuestro estudio a cinco tipos de datos escalares: **int**, **unsigned int**, **float**, **char** y **unsigned char**. Puedes consultar el resto de tipos escalares en el apéndice A. Encontrarás una variedad enorme: enteros con diferente número de bits, con y sin signo, flotantes de precisión normal y grande, booleanos, etc. Esa enorme variedad es uno de los puntos fuertes de C, pues permite ajustar el consumo de memoria a las necesidades de cada programa. En aras de la simplicidad expositiva, no obstante, no la consideraremos en el texto.

1.7.1. El tipo int

El tipo de datos **int** se usa normalmente para representar números enteros. La especificación de C no define el rango de valores que podemos representar con una variable de tipo **int**, es decir, no define el número de bits que ocupa una variable de tipo **int**. No obstante, lo más frecuente es que ocupe 32 bits. Nosotros asumiremos en este texto que el tamaño de un entero es de 32 bits, es decir, 4 bytes.

Como los enteros se codifican en complemento a 2, el rango de valores que podemos representar es $[-2147483648, 2147483647]$, es decir, $[-2^{31}, 2^{31}-1]$. Este rango es suficiente para las aplicaciones que presentaremos. Si resulta insuficiente o excesivo para alguno de tus programas, consulta el catálogo de tipos que presentamos en el apéndice A.

En C, tradicionalmente, los valores enteros se han utilizado para codificar valores booleanos. El valor 0 representa el valor lógico «falso» y cualquier otro valor representa «cierto». En la última revisión de C se ha introducido un tipo booleano, aunque no lo usaremos en este texto porque, de momento, no es frecuente encontrar programas que lo usen.

1.7.2. El tipo unsigned int

¿Para qué desperdiciar el bit más significativo en una variable entera de 32 bits que nunca almacenará valores negativos? C te permite definir variables de tipo «entero sin signo». El tipo tiene un nombre compuesto por dos palabras: «**unsigned int**» (aunque la palabra **unsigned**, sin más, es sinónimo de **unsigned int**).

Gracias al aprovechamiento del bit extra es posible aumentar el rango de valores positivos representables, que pasa a ser $[0, 2^{32} - 1]$, o sea, $[0, 4294967295]$.

1.7.3. El tipo float

El tipo de datos **float** representa números en coma flotante de 32 bits. La codificación de coma flotante permite definir valores con decimales. El máximo valor que puedes almacenar en una variable de tipo **float** es $3.40282347 \cdot 10^{38}$. Recuerda que el factor exponencial se codifica en los programas C con la letra «e» (o «E») seguida del exponente. Ese valor, pues, se codifica así en un programa C: 3.40282347e38. El número no nulo más pequeño (en valor absoluto) que puedes almacenar en una variable **float** es $1.17549435 \cdot 10^{-38}$ (o sea, el literal flotante 1.17549435e-38). Da la impresión, pues, de que podemos representar números con 8 decimales. No es así: la precisión no es la misma para todos los valores: es tanto mayor cuanto más próximo a cero es el valor.

1.7.4. El tipo char

El tipo **char**, aunque tenga un nombre que parezca sugerir el término «carácter» (que en inglés es «character») designa en realidad a una variante de enteros: el conjunto de números que podemos representar (en complemento a 2) con un solo byte (8 bits). El rango de valores que puede tomar una variable de tipo **char** es muy limitado: $[-128, 127]$.

Es frecuente usar variables de tipo **char** para almacenar caracteres (de ahí su nombre) codificados en ASCII o alguna de sus extensiones (como IsoLatin1). Si una variable *a* es de tipo **char**, la asignación *a*='0' es absolutamente equivalente a la asignación *a*=48, pues el valor ASCII del dígito 0 es 48.

1.7.5. El tipo unsigned char

Y del mismo modo que había una versión para enteros de 32 bits sin signo, hay una versión de **char** sin signo: **unsigned char**. Con un **unsigned char** se puede representar cualquier entero en el rango $[0, 255]$.

1.8. Se debe declarar el tipo de toda variable antes de usarla

Recuerda que en C *toda variable usada en un programa debe declararse antes de ser usada*. Declarar la variable consiste en darle un nombre (identificador) y asignarle un tipo.

1.8.1. Identificadores válidos

Las reglas para construir identificadores válidos son las mismas que sigue Python: un identificador es una sucesión de letras (del alfabeto inglés), dígitos y/o el carácter de subrayado (_) cuyo primer carácter no es un dígito. Y al igual que en Python, no puedes usar una palabra reservada como identificador. He aquí la relación de palabras reservadas del lenguaje C: **auto**, **break**, **case**, **char**, **const**, **continue**, **default**, **do**, **double**, **else**, **enum**, **extern**, **float**, **for**, **goto**, **if**, **int**, **long**, **register**, **return**, **short**, **signed**, **sizeof**, **static**, **struct**, **switch**, **typedef**, **union**, **unsigned**, **void**, **volatile** y **while**.

1.8.2. Sentencias de declaración

Una variable se declara precediendo su identificador con el tipo de datos de la variable. Este fragmento, por ejemplo, declara una variable de tipo entero, otra de tipo entero de un byte (o carácter) y otra de tipo flotante:

```
int a;
char b;
float c;
```

Se puede declarar una serie de variables del mismo tipo en una sola sentencia de declaración separando sus identificadores con comas. Este fragmento, por ejemplo, declara tres variables de tipo entero y otras dos de tipo flotante.

```
int x, y, z;
float u, v;
```

En `sumatorio.c` se declaran tres variables de tipo **int**, *a*, *b* y *i*, y una de tipo **float**, *s*.

Una variable declarada como de tipo entero sólo puede almacenar valores de tipo entero. *Una vez se ha declarado una variable, es imposible cambiar su tipo*, ni siquiera volviendo a declararla. Este programa, por ejemplo, es incorrecto por el intento de redeclarar el tipo de la variable *a*:

C, ocupación de los datos, complemento a 2 y portabilidad

Los números enteros con signo se codifican en complemento a 2. Con n bits puedes representar valores enteros en el rango $[-2^{n-1}, 2^{n-1}-1]$. Los valores positivos se representan en binario, sin más. Los valores negativos se codifican representando en binario su valor absoluto, invirtiendo todos sus bits y añadiendo 1 al resultado. Supón que trabajamos con datos de tipo **char** (8 bits). El valor 28 se representa en binario así 00011100. El valor -28 se obtiene tomando la representación binaria de 28, invirtiendo sus bits (11100011), y añadiendo uno. El resultado es 11100100.

Una ventaja de la notación en complemento a 2 es que simplifica el diseño de circuitos para la realización de cálculos aritméticos. Por ejemplo, la resta es una simple suma. Si deseas restar a 30 el valor 28, basta con sumar 30 y -28 con la misma circuitería electrónica utilizada para efectuar sumas convencionales:

```
  00011110
+ 11100100
----------
  00000010
```

El complemento a 2 puede gastarte malas pasadas si no eres consciente de cómo funciona. Por ejemplo, sumar dos números positivos puede producir un resultado ¡negativo! Si trabajas con 8 bits y sumas 127 y 1, obtienes el valor -128:

```
  01111111
+ 00000001
----------
  10000000
```

Este fenómeno se conoce como «desbordamiento». C no aborta la ejecución del programa cuando se produce un desbordamiento: da por bueno el resultado y sigue. Mala cosa: puede que demos por bueno un programa que está produciendo resultados erróneos.

El estándar de C no define de modo claro la ocupación de cada uno de sus tipos de datos lo cual, unido a fenómenos de desbordamiento, dificulta notablemente la portabilidad de los programas. En la mayoría de los compiladores y ordenadores actuales, una variable de tipo **int** ocupa 32 bits. Sin embargo, en ordenadores más antiguos era frecuente que ocupara sólo 16. Un programa que suponga una representación mayor que la real puede resultar en la comisión de errores en tiempo de ejecución. Por ejemplo, si una variable a de tipo **int** ocupa 32 bits y vale 32767, ejecutar la asignación $a = a + 1$ almacenará en a el valor 32768; pero si el tipo **int** ocupa 16 bits, se almacena el valor -32768.

Puede que demos por bueno un programa al compilarlo y ejecutarlo en una plataforma determinada, pero que falle estrepitosamente cuando lo compilamos y ejecutamos en una plataforma diferente. O, peor aún, puede que el error pase inadvertido durante mucho tiempo: el programa no abortará la ejecución y producirá resultados incorrectos que podemos no detectar. Es un problema muy grave.

Los problemas relacionados con la garantía de poder ejecutar un mismo programa en diferentes plataformas se conocen como problemas de portabilidad. Pese a los muchos problemas de portabilidad de C, es el lenguaje de programación en el que se ha escrito buena parte de los programas que hoy ejecutamos en una gran variedad de plataformas.

redeclara.c ⚡ redeclara.c ⚡

```c
#include <stdio.h>

int main(void)
{
  int a;
  float a;

```

> ***char, unsigned char, ASCII e IsoLatin1***
>
> La tabla ASCII tiene caracteres asociados a valores comprendidos entre 0 y 127, así que todo carácter ASCII puede almacenarse en una variable de tipo **char**. Pero, en realidad, nosotros no usamos la tabla ASCII «pura», sino una extensión suya: IsoLatin1 (también conocida por ISO-8859-1 o ISO-8859-15, si incluye el símbolo del euro). La tabla IsoLatin1 nos permite utilizar caracteres acentuados y otros símbolos especiales propios de las lenguas románicas occidentales. ¿Qué ocurre si asignamos a una variable de tipo **char** el carácter `'á'`? El código IsoLatin1 de `'á'` es 225, que es un valor numérico mayor que 127, el máximo valor entero que podemos almacenar en una variable de tipo **char**. Mmmm. Sí, pero 225 se codifica en binario como esta secuencia de ceros y unos: `11100001`. Si interpretamos dicha secuencia en complemento a dos, tenemos el valor −31, y ese es, precisamente, el valor que resulta almacenado. Podemos evitar este inconveniente usando el tipo **unsigned char**, pues permite almacenar valores entre 0 y 255.

```
8     a = 2;
9     return 0;
10 }
```

Al compilarlo obtenemos este mensaje de error:

```
$ gcc redeclara.c -o redeclara ↲
redeclara.c: In function ‘main’:
redeclara.c:6: conflicting types for ‘a’
redeclara.c:5: previous declaration of ‘a’
```

El compilador nos indica que la variable *a* presenta un conflicto de tipos en la línea 6 y que ya había sido declarada previamente en la línea 5.

1.8.3. Declaración con inicialización

Debes tener presente que el valor inicial de una variable declarada está indefinido. *Jamás debes acceder al contenido de una variable que no haya sido previamente inicializada.* Si lo haces, el compilador no detectará error alguno, pero tu programa presentará un comportamiento indeterminado: a veces funcionará bien, y a veces mal, lo cual es peor que un funcionamiento siempre incorrecto, pues podrías llegar a dar por bueno un programa mal escrito. En esto C se diferencia de Python: Python abortaba la ejecución de un programa cuando se intentaba usar una variable no inicializada; C no aborta la ejecución, pero presenta un comportamiento indeterminado.

Puedes inicializar las variables en el momento de su declaración. Para ello, basta con añadir el operador de asignación y un valor a continuación de la variable en cuestión.

Mira este ejemplo:

```
#include <stdio.h>

int main(void)
{
  int a = 2;
  float b = 2.0, c, d = 1.0, e;

  return 0;
}
```

En él, las variables *a*, *b* y *d* se inicializan en la declaración y las variables *c* y *e* no tienen valor definido al ser declaradas.

Recuerda que acceder a variables no inicializadas es una fuente de graves errores. Acostúmbrate a inicializar las variables tan pronto puedas.

1.9. Salida por pantalla

La función de impresión de información en pantalla utilizada habitualmente es *printf*. Es una función disponible al incluir *stdio.h* en el programa. El uso de *printf* es ligeramente más complicado que el de la sentencia *print* de Python, aunque no te resultará difícil si ya has aprendido a utilizar el operador de formato en Python (%).

En su forma de uso más simple, *printf* permite mostrar una cadena por pantalla.

```
#include <stdio.h>

int main(void)
{
  printf("Una␣cadena");
  printf("y␣otra.");
  return 0;
}
```

La función *printf* *no* añade un salto de línea automáticamente, como sí hacía *print* en Python. En el programa anterior, ambas cadenas se muestran una a continuación de otra. Si deseas que haya un salto de línea, deberás escribir `\n` al final de la cadena.

```
#include <stdio.h>

int main(void)
{
  printf("Una␣cadena\n");
  printf("y␣otra.\n");
  return 0;
}
```

1.9.1. Marcas de formato para la impresión de valores con *printf*

Marcas de formato para números

Para mostrar números enteros o flotantes has de usar necesariamente cadenas con formato. Afortunadamente, las marcas que aprendiste al estudiar Python se utilizan en C. Eso sí, hay algunas que no te hemos presentado aún y que también se recogen en esta tabla:

Tipo	Marca
int	%d
unsigned int	%u
float	%f
char	%hhd
unsigned char	%hhu

Por ejemplo, si *a* es una variable de tipo **int** con valor 5, *b* es una variable de tipo **float** con valor 1.0, y *c* es una variable de tipo **char** con valor 100, esta llamada a la función *printf*:

```
  printf("Un␣entero:␣%d,␣un␣flotante:␣%f,␣un␣byte:␣%hhd\n", a, b, c);
```

muestra por pantalla esto:

```
Un entero: 5, un flotante: 1.000000, un byte: 100
```

¡Ojo! a la cadena de formato le sigue una coma, y no un operador de formato como sucedía en Python. Cada variable se separa de las otras con una coma.

. . . . EJERCICIOS . . . .

▶ 15 ¿Que mostrará por pantalla esta llamada a *printf* suponiendo que *a* es de tipo entero y vale 10?

```
printf("%d-%d\n", a+1, 2+2);
```

. . . .

Las marcas de formato para enteros aceptan *modificadores*, es decir, puedes alterar la representación introduciendo ciertos caracteres entre el símbolo de porcentaje y el resto de la marca. Aquí tienes los principales:

- Un número positivo: reserva un número de espacios determinado (el que se indique) para representar el valor y muestra el entero alineado a la derecha.

  Ejemplo: la sentencia

  ```
  printf("[%6d]", 10);
  ```

  muestra en pantalla:

  ```
  [␣␣␣␣10]
  ```

- Un número negativo: reserva tantos espacios como indique el valor absoluto del número para representar el entero y muestra el valor alineado a la izquierda.

  Ejemplo: la sentencia

  ```
  printf("[%-6d]", 10);
  ```

  muestra en pantalla:

  ```
  [10␣␣␣␣]
  ```

- Un número que empieza por cero: reserva tantos espacios como indique el número para representar el entero y muestra el valor alineado a la derecha. Los espacios que no ocupa el entero se rellenan con ceros.

  Ejemplo: la sentencia

  ```
  printf("[%06d]", 10);
  ```

  muestra en pantalla:

  ```
  [000010]
  ```

- El signo +: muestra explícitamente el signo (positivo o negativo) del entero.

  Ejemplo: la sentencia

  ```
  printf("[%+6d]", 10);
  ```

  muestra en pantalla:

  ```
  [␣␣␣+10]
  ```

Hay dos notaciones alternativas para la representación de flotantes que podemos seleccionar mediante la marca de formato adecuada:

Tipo	Notación	Marca
float	Convencional	%f
float	Científica	%e

La forma convencional muestra los números con una parte entera y una decimal separadas por un punto. La notación científica representa al número como una cantidad con una sola cifra entera y una parte decimal, pero seguida de la letra «e» y un valor entero. Por ejemplo, en notación científica, el número 10.1 se representa con 1.010000e+01 y se interpreta así: 1.01×10^1.

También puedes usar modificadores para controlar la representación en pantalla de los flotantes. Los modificadores que hemos presentado para los enteros son válidos aquí. Tienes, además, la posibilidad de fijar la precisión:

- Un punto seguido de un número: indica cuántos decimales se mostrarán.

  Ejemplo: la sentencia

  ```
  printf("[%6.2f]", 10.1);
  ```

  muestra en pantalla:

  ```
  [␣10.10]
  ```

Marcas de formato para texto

Y aún nos queda presentar las marcas de formato para texto. C distingue entre caracteres y cadenas:

Tipo	Marca
carácter	%c
cadena	%s

¡Atención! La marca `%c` muestra como carácter un número entero. Naturalmente, el carácter que se muestra es el que corresponde al valor entero según la tabla ASCII (o, en tu ordenador, IsoLatin1 si el número es mayor que 127). Por ejemplo, la sentencia

```
printf("[%c]", 97);
```

muestra en pantalla:

```
[a]
```

Recuerda que el valor 97 también puede representarse con el literal `'a'`, así que esta otra sentencia

```
printf("[%c]", 'a');
```

también muestra en pantalla esto:

```
[a]
```

Aún no sabemos almacenar cadenas en variables, así que poca aplicación podemos encontrar de momento a la marca `%s`. He aquí, de todos modos, un ejemplo trivial de uso:

```
printf("[%s]", "una␣cadena");
```

En pantalla se muestra esto:

```
[una cadena]
```

También puedes usar números positivos y negativos como modificadores de estas marcas. Su efecto es reservar los espacios que indiques y alinear a derecha o izquierda.

Aquí tienes un programa de ejemplo en el que se utilizan diferentes marcas de formato con y sin modificadores.

modificadores.c

```c
#include <stdio.h>

int main(void)
{
  char c = 'a';
  int i = 1000000;
  float f = 2e1;

  printf("c   : %c %hhd  <- ¡IMPORTANTE! Estudia la diferencia.\n", c, c);
  printf("i   : %d |%10d|%-10d|\n", i, i, i);
  printf("f   : %f |%10.2f|%+4.2f|\n", f, f, f);
  return 0;
}
```

El resultado de ejecutar el programa es la impresión por pantalla del siguiente texto:

```
c   : a 97  <- ¡IMPORTANTE! Estudia la diferencia.
i   : 1000000 |   1000000|1000000   |
f   : 20.000000 |     20.00|+20.00|
```

EJERCICIOS

▶ **16** ¿Qué muestra por pantalla cada uno de estos programas?

a) ascii1.c

```c
#include <stdio.h>

int main(void)
{
  char i;
  for (i='A'; i<='Z'; i++)
    printf("%c", i);
  printf("\n");
  return 0;
}
```

b) ascii2.c

```c
#include <stdio.h>

int main(void)
{
  char i;
  for (i=65; i<=90; i++)
    printf("%c", i);
  printf("\n");
  return 0;
}
```

c) ascii3.c

```c
#include <stdio.h>
```

```

int main(void)
{
  int i;
  for (i='A'; i<='Z'; i++)
    printf("%d ", i);
  printf("\n");
  return 0;
}
```

d) ascii4.c

```
#include <stdio.h>

int main(void)
{
  int i;
  for (i='A'; i<='Z'; i++)
    printf("%d-%c ", i, i);
  printf("\n");
  return 0;
}
```

e) ascii5.c

```
#include <stdio.h>

int main(void)
{
  char i;
  for (i='A'; i<='z'; i++) // Ojo: la z es minúscula.
    printf("%d ", (int) i);
  printf("\n");
  return 0;
}
```

▶ 17 Diseña un programa que muestre la tabla ASCII desde su elemento de código numérico 32 hasta el de código numérico 126. En la tabla se mostrarán los códigos ASCII, además de las respectivas representaciones como caracteres de sus elementos. Aquí tienes las primeras y últimas líneas de la tabla que debes mostrar (debes hacer que tu programa muestre la información exactamente como se muestra aquí):

```
+---------+----------+
| Decimal | Carácter |
+---------+----------+
|      32 |          |
|      33 |        ! |
|      34 |        " |
|      35 |        # |
|      36 |        $ |
|      37 |        % |

      ...       ...

|     124 |        | |
|     125 |        } |
|     126 |        ~ |
+---------+----------+
```

. . . .

Hay un rico juego de marcas de formato y las recogemos en el apéndice A. Consúltalo si usas tipos diferentes de los que presentamos en el texto o si quieres mostrar valores enteros en base 8 o 16. En cualquier caso, es probable que necesites conocer una marca especial, `%%`, que sirve para mostrar el símbolo de porcentaje. Por ejemplo, la sentencia

```
printf("[%d%%]", 100);
```

muestra en pantalla:

```
[100%]
```

1.10. Variables y direcciones de memoria

Antes de presentar con cierto detalle la entrada de datos por teclado mediante *scanf*, nos conviene detenernos brevemente para estudiar algunas cuestiones relativas a las variables y la memoria que ocupan.

Recuerda que la memoria es una sucesión de celdas numeradas y que una dirección de memoria no es más que un número entero. La declaración de una variable supone la reserva de una zona de memoria lo suficientemente grande para albergar su contenido. Cuando declaramos una variable de tipo **int**, por ejemplo, se reservan 4 bytes de memoria en los que se almacenará (codificado en complemento a 2) el valor de dicha variable. Modificar el valor de la variable mediante una asignación supone modificar el patrón de 32 bits (4 bytes) que hay en esa zona de memoria.

Este programa, por ejemplo,

```
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 0;
  b = a + 8;

  return 0;
}
```

reserva 8 bytes para albergar dos valores enteros.[9] Imagina que *a* ocupa los bytes 1000–1003 y *b* ocupa los bytes 1004–1007. Podemos representar la memoria así:

996:	01010010	10101000	01110011	11110010	
1000:	01011010	00111101	00111010	11010111	*a*
1004:	10111011	10010110	01010010	01010011	*b*
1008:	11010111	01000110	11110010	01011101	

Observa que, inicialmente, cuando se reserva la memoria, ésta contiene un patrón de bits arbitrario. La sentencia *a* = 0 se interpreta como «almacena el valor 0 en la dirección de memoria de *a*», es decir, «almacena el valor 0 en la dirección de memoria 1000»[10]. Este es el resultado de ejecutar esa sentencia:

[9] En el apartado 3.5.2 veremos que la reserva se produce en una zona de memoria especial llamada pila. No conviene que nos detengamos ahora a considerar los matices que ello introduce en el discurso.

[10] En realidad, en la zona de memoria 1000–1003, pues se modifica el contenido de 4 bytes. En aras de la brevedad, nos referiremos a los 4 bytes sólo con la dirección del primero de ellos.

996:	01010010	10101000	01110011	11110010	
1000:	00000000	00000000	00000000	00000000	a
1004:	10111011	10010110	01010010	01010011	b
1008:	11010111	01000110	11110010	01011101	

La asignación *b* = *a* + 8 se interpreta como «calcula el valor que resulta de sumar 8 al contenido de la dirección de memoria 1000 y deja el resultado en la dirección de memoria 1004».

996:	01010010	10101000	01110011	11110010	
1000:	00000000	00000000	00000000	00000000	a
1004:	00000000	00000000	00000000	00001000	b
1008:	11010111	01000110	11110010	01011101	

Hemos supuesto que *a* está en la dirección 1000 y *b* en la 1004, pero ¿podemos saber en qué direcciones de memoria se almacenan realmente *a* y *b*? Sí: el operador & permite conocer la dirección de memoria en la que se almacena una variable:

direcciones.c **direcciones.c**

```c
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 0;
  b = a + 8;

  printf("Dirección de a: %u\n", (unsigned int)&a);
  printf("Dirección de b: %u\n", (unsigned int)&b);

  return 0;
}
```

Observa que usamos la marca de formato %u para mostrar el valor de la dirección de memoria, pues debe mostrarse como entero sin signo. La conversión a tipo **unsigned int** evita molestos mensajes de aviso al compilar.[11]

Al ejecutar el programa tenemos en pantalla el siguiente texto (puede que si ejecutas tú mismo el programa obtengas un resultado diferente):

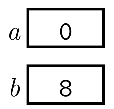
```
Dirección de a: 3221222580
Dirección de b: 3221222576
```

O sea, que en realidad este otro gráfico representa mejor la disposición de las variables en memoria:

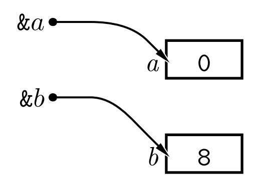
3221222572:	01010010	10101000	01110011	11110010	
3221222576:	00000000	00000000	00000000	00001000	b
3221222580:	00000000	00000000	00000000	00000000	a
3221222584:	11010111	01000110	11110010	01011101	

[11] Hay una marca especial, %p, que muestra directamente la dirección de memoria sin necesidad de efectuar la conversión a **unsigned int**, pero lo hace usando notación hexadecimal.

Normalmente no necesitamos saber en qué dirección de memoria se almacena una variable, así que no recurriremos a representaciones gráficas tan detalladas como las que hemos presentado. Usualmente nos conformaremos con representar las variables escalares mediante cajas y representaremos su valor de una forma más cómodamente legible que como una secuencia de bits. La representación anterior se simplificará, pues, así:



Las direcciones de memoria de las variables se representarán con flechas que apuntan a sus correspondientes cajas:



Ahora que hemos averiguado nuevas cosas acerca de las variables, vale la pena que reflexionemos brevemente sobre el significado de los identificadores de variables allí donde aparecen. Considera este sencillo programa:

```
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 0;
  b = a;
  scanf("%d", &b);
  a = a + b;

  return 0;
}
```

¿Cómo se interpreta la sentencia de asignación *a* = 0? Se interpreta como «almacena el valor 0 en la dirección de memoria de *a*». ¿Y *b* = *a*?, ¿cómo se interpreta? Como «almacena una copia del contenido de *a* en la dirección de memoria de *b*». Fíjate bien, el identificador *a* recibe interpretaciones diferentes según aparezca a la izquierda o a la derecha de una asignación:

- a la izquierda del igual, significa «la dirección de *a*»,
- y a la derecha, es decir, en una expresión, significa «el contenido de *a*».

La función *scanf* necesita una dirección de memoria para saber dónde debe depositar un resultado. Como no estamos en una sentencia de asignación, sino en una expresión, es necesario que obtengamos explícitamente la dirección de memoria con el operador &*b*. Así, para leer por teclado el valor de *b* usamos la llamada *scanf*("%d", &*b*).

EJERCICIOS

▶ **18** Interpreta el significado de la sentencia *a* = *a* + *b*.

1.11. Entrada por teclado

La función *scanf*, disponible al incluir `stdio.h`, permite leer datos por teclado. La función *scanf* se usa de un modo similar a *printf*: su primer argumento es una cadena con marcas de formato. A éste le siguen una o más direcciones de memoria. Si deseas leer por teclado el valor de una variable entera *a*, puedes hacerlo así:

```
scanf("%d", &a);
```

Observa que la variable cuyo valor se lee por teclado va *obligatoriamente* precedida por el operador `&`: es así como obtenemos la dirección de memoria en la que se almacena el valor de la variable. *Uno de los errores que cometerás con mayor frecuencia es omitir el carácter* `&` *que debe preceder a todas las variables escalares en scanf*.

Recuerda: la función *scanf* recibe estos datos:

- Una cadena cuya marca de formato indica de qué tipo es el valor que vamos a leer por teclado:

Tipo	Marca
int	`%d`
unsigned int	`%u`
float	`%f`
char como entero	`%hhd`
char como carácter	`%c`
unsigned char como entero	`%hhu`
unsigned char como carácter	`%c`

- La dirección de memoria que corresponde al lugar en el que se depositará el valor leído. Debemos proporcionar una dirección de memoria por cada marca de formato indicada en el primer argumento.

Observa que hay dos formas de leer un dato de tipo **char** o **unsigned char**: como entero (de un byte con o sin signo, respectivamente) o como carácter. En el segundo caso, se espera que el usuario teclee un solo carácter y se almacenará en la variable su valor numérico según la tabla ASCII o su extensión IsoLatin.

Una advertencia: la lectura de teclado en C presenta numerosas dificultades prácticas. Es muy recomendable que leas el apéndice B antes de seguir estudiando y *absolutamente necesario* que lo leas antes de empezar a practicar con el ordenador. Si no lo haces, muchos de tus programas presentarán un comportamiento muy extraño y no entenderás por qué. Tú mismo.

1.12. Expresiones

Muchos de los símbolos que representan a los operadores de Python que ya conoces son los mismos en C. Los presentamos ahora agrupados por familias. (Consulta los niveles de precedencia y asociatividad en la tabla de la página 44.) Presta especial atención a los operadores que no conoces por el lenguaje de programación Python, como son los operadores de bits, el operador condicional o los de incremento/decremento.

Operadores aritméticos Suma (`+`), resta (`-`), producto (`*`), división (`/`), módulo o resto de la división (`%`), identidad (`+` unario), cambio de signo (`-` unario).

No hay operador de exponenciación.[12]

[12]Pero hay una función de la biblioteca matemática que permite calcular la potencia de un número: *pow*.

Errores frecuentes en el uso de scanf

Es responsabilidad del programador pasar correctamente los datos a *scanf*. Un error que puede tener graves consecuencias consiste en pasar incorrectamente la dirección de memoria en la que dejará el valor leído. Este programa, por ejemplo, es erróneo:

```
scanf("%d", a);
```

La función *scanf* no está recibiendo *la dirección de memoria* en la que «reside» *a*, sino *el valor* almacenado en *a*. Si *scanf* interpreta dicho valor como una dirección de memoria (cosa que hace), guardará en ella el número que lea de teclado. ¡Y el compilador no necesariamente detectará el error! El resultado es catastrófico.

Otro error típico al usar *scanf* consiste en confundir el tipo de una variable y/o la marca de formato que le corresponde. Por ejemplo, imagina que *c* es una variable de tipo **char**. Este intento de lectura de su valor por teclado es erróneo:

```
scanf("%d", &c);
```

A *scanf* le estamos pasando la dirección de memoria de la variable *c*. Hasta ahí, bien. Pero *c* sólo ocupa un byte y a *scanf* le estamos diciendo que «rellene» 4 bytes con un número entero a partir de esa dirección de memoria. Otro error de consecuencias gravísimas. La marca de formato adecuada para leer un número de tipo **char** hubiera sido `%hhd`.

```
scanf("%hhd", &c);
```

La división de dos números enteros proporciona un resultado de tipo entero (como ocurría en Python).

Los operadores aritméticos sólo funcionan con datos numéricos[13]. No es posible, por ejemplo, concatenar cadenas con el operador + (cosa que sí podíamos hacer en Python).

La dualidad carácter-entero del tipo **char** hace que puedas utilizar la suma o la resta (o cualquier otro operador aritmético) con variables o valores de tipo **char**. Por ejemplo `'a'` + 1 es una expresión válida y su valor es `'b'` (o, equivalentemente, el valor 98, ya que `'a'` equivale a 97). (Recuerda, no obstante, que un carácter no es una cadena en C, así que `"a"` + 1 *no* es `"b"`.)

Operadores lógicos Negación o no-lógica (`!`), y-lógica o conjunción (&&) y o-lógica o disyunción (||).

Los símbolos son diferentes de los que aprendimos en Python. La negación era allí **not**, la conjunción era **and** y la disyunción **or**.

C sigue el convenio de que 0 significa *falso* y cualquier otro valor significa *cierto*. Así pues, cualquier valor entero puede interpretarse como un valor lógico, igual que en Python.

Operadores de comparación Igual que (==), distinto de (!=), menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=). Son viejos conocidos. Una diferencia con respecto a Python: sólo puedes usarlos para comparar valores escalares. No puedes, por ejemplo, comparar cadenas mediante estos operadores.

[13] Y la suma y la resta trabajan también con punteros. Ya estudiaremos la denominada «aritmética de punteros» más adelante.

-*Wall*

Cuando escribimos un texto en castellano podemos cometer tres tipos de errores:

- Errores léxicos: escribimos palabras incorrectamente, con errores ortográficos, o usamos palabras inexistentes. Por ejemplo: «`herror`», «`lécsico`», «`jerigóndor`».
- Errores sintácticos: las palabras son válidas y están bien escritas, pero faltan componentes de una frase (como el sujeto o el verbo), los componentes no concuerdad o no ocupan la posición adecuada, etc. Por ejemplo: «`el error sintáctica son`», «`la compilador detectó errores`».
- Errores semánticos: la frase está correctamente construida pero carece de significado válido en el lenguaje. Por ejemplo: «`el compilador silbó una tonada en vídeo`», «`los osos son enteros con decimales romos`».

Lo mismo ocurre con los programas C; pueden contener errores de los tres tipos:

- Errores léxicos: usamos carácteres no válidos o construimos incorrectamente componentes elementales del programa (como identificadores, cadenas, palabras clave, etc.). Por ejemplo: «`@3`», «`"una cadena sin cerrar`».
- Errores sintácticos: construimos mal una sentencia aunque usamos palabras válidas. Por ejemplo: «`while a < 10 { a += 1; }`», «`b = 2 * / 3;`».
- Errores semánticos: la sentencia no tiene un significado válido. Por ejemplo, si *a* es de tipo **float**, estas sentencias contienen errores semánticos: «`scanf("%d", &a);`» (no se puede leer *a* como entero), «`if (a = 1.0) { a = 2.0; }`» (no compara el valor de *a* con 1.0: le asigna 1.0 a *a*).

El compilador de C no deja pasar errores léxicos o sintácticos: informa y no traduce el programa a código de máquina. Con los errores semánticos, sin embargo, el compilador es más indulgente: la filosofía de C es suponer que el programador puede tener una razón para hacer lo que expresa en los programas, aunque no tenga un significado «correcto» a primera vista. No obstante, y para según qué posibles errores, el compilador puede emitir avisos (*warnings*). El nivel de avisos es configurable. La opción `-Wall` («Warning all») activa todos los avisos, lo que incluye algunos potenciales errores semánticos. Este programa erróneo, por ejemplo, no genera ningún aviso al compilarse sin -*Wall*:

semanticos.c ↯ semanticos.c ↯

```
1 #include <stdio.h>
2 int main(void)
3 {
4   float a;
5   scanf("%d", &a);
6   if (a = 0.0) { a = 2.0; }
7   return 0;
8 }
```

Pero si lo compilas con «`gcc -Wall semanticos.c -o semanticos`»:

```
$ gcc -Wall semanticos.c -o semanticos ↵
semanticos.c: In function ‘main’:
semanticos.c:5: warning: int format, float arg (arg 2)
semanticos.c:6: warning: suggest parentheses around assignment used
as truth value
```

El compilador advierte de posibles errores semánticos en las líneas 5 y 6. Necesitarás práctica para descifrar mensajes tan parcos o extraños como los que produce `gcc`, así que acostúmbrate a compilar con `-Wall`. (Y hazlo *siempre* que tu programa presente un comportamiento anómalo en ejecución.)

Lecturas múltiples con scanf

No te hemos contado todo sobre *scanf*. Puedes usar *scanf* para leer más de un valor. Por ejemplo, este programa lee dos valores enteros con un solo *scanf*:

lectura_multiple.c   lectura_multiple.c

```c
#include <stdio.h>

int main(void)
{
  int a, b;
  printf("Introduce dos enteros: ");
  scanf("%d %d", &a, &b);
  printf("Valores leídos: %d y %d\n", a, b);
  return 0;
}
```

También podemos especificar con cierto detalle cómo esperamos que el usuario introduzca la información. Por ejemplo, con `scanf("%d-%d", &a, &b)` indicamos que el usuario debe separar los enteros con un guión; y con `scanf("(%d,%d)", &a, &b)` especificamos que esperamos encontrar los enteros encerrados entre paréntesis y separados por comas.

Lee la página de manual de *scanf* (escribiendo `man 3 scanf` en el intérprete de órdenes Unix) para obtener más información.

La evaluación de una comparación proporciona un valor entero: 0 si el resultado es falso y cualquier otro si el resultado es cierto (aunque normalmente el valor para cierto es 1).

Operadores de bits Complemento (`~`), «y» (`&`), «o» (`|`), «o» exclusiva (`^`), desplazamiento a izquierdas (`<<`), desplazamiento a derechas (`>>`). Estos operadores trabajan directamente con los bits que codifican un valor entero. Aunque también están disponibles en Python, no los estudiamos entonces porque son de uso infrecuente en ese lenguaje de programación.

El operador de complemento es unario e invierte todos los bits del valor. Tanto `&` como `|` y `^` son operadores binarios. El operador `&` devuelve un valor cuyo *n*-ésimo bit es 1 si y sólo si los dos bits de la *n*-ésima posición de los operandos son también 1. El operador `|` devuelve 0 en un bit si y solo si los correspondientes bits en los operandos son también 0. El operador `^` devuelve 1 si y sólo si los correspondientes bits en los operandos son diferentes. Lo entenderás mejor con un ejemplo. Imagina que *a* y *b* son variables de tipo **char** que valen 6 y 3, respectivamente. En binario, el valor de *a* se codifica como 00000110 y el valor de *b* como 00000011. El resultado de *a* `|` *b* es 7, que corresponde al valor en base diez del número binario 000000111. El resultado de *a* `&` *b* es, en binario, 000000010, es decir, el valor decimal 2. El resultado binario de *a* `^` *b* es 000000101, que en base 10 es 5. Finalmente, el resultado de `~`*a* es 11111001, es decir, −7 (recuerda que un número con signo está codificado en complemento a 2, así que si su primer bit es 1, el número es negativo).

Los operadores de desplazamiento desplazan los bits un número dado de posiciones a izquierda o derecha. Por ejemplo, 16 como valor de tipo **char** es 00010000, así que 16 << 1 es 32, que en binario es 00100000, y 16 >> 1 es 8, que en binario es 00001000.

Operadores de asignación Asignación (`=`), asignación con suma (`+=`), asignación con resta (`-=`), asignación con producto (`*=`), asignación con división (`/=`), asignación con

> ***Operadores de bits y programación de sistemas***
>
> C presenta una enorme colección de operadores, pero quizá los que te resulten más llamativos sean los operadores de bits. Difícilmente los utilizarás en programas convencionales, pero son insustituibles en la programación de sistemas. Cuando manejes información a muy bajo nivel es probable que necesites acceder a bits y modificar sus valores.
>
> Por ejemplo, el control de ciertos puertos del ordenador pasa por leer y asignar valores concretos a ciertos bits de direcciones virtuales de memoria. Puede que poner a 1 el bit menos significativo de determinada dirección permita detener la actividad de una impresora conectada a un puerto paralelo, o que el bit más significativo nos alerte de si falta papel en la impresora.
>
> Si deseas saber si un bit está o no activo, puedes utilizar los operadores & y <<. Para saber, por ejemplo, si el octavo bit de una variable *x* está activo, puedes calcular *x* & (1 << 7). Si el resultado es cero, el bit no está activo; en caso contrario, está activo. Para fijar a 1 el valor de ese mismo bit, puedes hacer *x* = *x* | (1 << 7).
>
> Los operadores de bits emulan el comportamiento de ciertas instrucciones disponibles en los lenguajes ensambladores. La facilidad que proporciona C para escribir programas de «bajo nivel» es grande, y por ello C se considera el lenguaje a elegir cuando hemos de escribir un controlador para un dispositivo o el código de un sistema operativo.

módulo (%=), asignación con desplazamiento a izquierda (<<=), asignación con desplazamiento a derecha (>>=), asignación con «y» (&=), asignación con «o» (|=), asignación con «o» exclusiva (^=).

Puede resultarte extraño que la asignación se considere también un operador. Que sea un operador permite escribir asignaciones múltiples como ésta:

```
a = b = 1;
```

Es un operador asociativo por la derecha, así que las asignaciones se ejecutan en este orden:

```
a = (b = 1);
```

El valor que resulta de evaluar una asignación con = es el valor asignado a su parte izquierda. Cuando se ejecuta *b* = 1, el valor asignado a *b* es 1, así que ese valor es el que se asigna también a *a*.

La asignación con una operación «op» hace que a la variable de la izquierda se le asigne el resultado de operar con «op» su valor con el operando derecho. Por ejemplo, *a* /= 3 es equivalente a *a* = *a* / 3.

Este tipo de asignación con operación recibe el nombre de *asignación aumentada*.

Operador de tamaño sizeof.

El operador **sizeof** puede aplicarse a un nombre de tipo (encerrado entre paréntesis) o a un identificador de variable. En el primer caso devuelve el número de bytes que ocupa en memoria una variable de ese tipo, y en el segundo, el número de bytes que ocupa esa variable. Si *a* es una variable de tipo **char**, tanto **sizeof**(*a*) como **sizeof**(**char**) devuelven el valor 1. Ojo: recuerda que 'a' es literal entero, así que **sizeof**('a') vale 4.

Operadores de coerción o conversión de tipos (en inglés «type casting operator»). Puedes convertir un valor de un tipo de datos a otro que sea «compatible». Para ello dispones de operadores de la forma (**tipo**), donde **tipo** es **int**, **float**, etc.

Por ejemplo, si deseas efectuar una división entre enteros que no pierda decimales al convertir el resultado a un flotante, puedes hacerlo como te muestra este programa:

```
#include <stdio.h>

int main(void)
{
  float x;
  int a = 1, b = 2;

  x = a / (float) b;
}
```

En este ejemplo, hemos convertido el valor de *b* a un **float** antes de efectuar la división. Es similar a la función *float* de Python, sólo que en Python se hacía la conversión con una llamada a función como *float*(*b*), y aquí utilizamos un operador prefijo: (**float**) *b*. Es una notación bastante extraña, así que es probable que te confunda durante un tiempo. En la siguiente sección abundaremos en la conversión de tipos en C.

Operador condicional (?:).

Este operador no tiene correlato en Python. Hay tres operandos: una condición y dos expresiones[14]. El resultado de la operación es el valor de la primera expresión si la condición es cierta y el valor de la segunda si es falsa. Por ejemplo, la asignación

```
a = (x > 10) ? 100 : 200
```

almacena en *a* el valor 100 o 200, dependiendo de si *x* es o no es mayor que 10. Es equivalente a este fragmento de programa:

```
if (x > 10)
  a = 100;
else
  a = 200;
```

Operadores de incremento/decremento Preincremento (++ en forma prefija), postincremento (++ en forma postfija), predecremento (-- en forma prefija), postdecremento (-- en forma postfija).

Estos operadores no tienen equivalente inmediato en Python. Los operadores de incremento y decremento pueden ir delante de una variable (forma prefija) o detrás (forma postfija). En ambos casos incrementan (++) o decrementan (--) en una unidad el valor de la variable entera.

Si *i* vale 1, valdrá 2 después de ejecutar ++*i* o *i*++, y valdrá 0 después de ejecutar --*i* o *i*--. Hay una diferencia importante entre aplicar estos operadores en forma prefija o sufija.

- La expresión *i*++ *primero* se evalúa como el valor actual de *i* y *después* hace que *i* incremente su valor en una unidad.
- La expresión ++*i* *primero* incrementa el valor de *i* en una unidad y *después* se evalúa como el valor actual (que es el que resulta de efectuar el incremento).

[14] Lo cierto es que hay tres expresiones, pues la comparación no es más que una expresión. Si dicha expresión devuelve el valor 0, se interpreta el resultado como «falso»; en caso contrario, el resultado es «cierto».

Si el operador se está aplicando en una expresión, esta diferencia tiene importancia. Supongamos que *i* vale 1 y que evaluamos esta asignación:

```
a = i++;
```

La variable *a* acaba valiendo 1 e *i* acaba valiendo 2. Fíjate: al ser un *post*incremento, primero se devuelve el valor de *i*, que se asigna a *a*, y después se incrementa *i*.

Al ejecutar esta otra asignación obtenemos un resultado diferente:

```
a = ++i;
```

Tanto *a* como *i* acaban valiendo 2. El operador de *pre*incremento primero asigna a *i* su valor actual incrementado en una unidad y después devuelve ese valor (ya incrementado), que es lo que finalmente estamos asignando a *a*.

Lo mismo ocurre con los operadores de pre y postdecremento, pero, naturalmente, decrementado el valor en una unidad en lugar de incrementarlo.

Que haya operadores de pre y postincremento (y pre y postdecremento) te debe parecer una rareza excesiva y pensarás que nunca necesitarás hilar tan fino. Si es así, te equivocas: en los próximos capítulos usaremos operadores de incremento y necesitaremos escoger entre *pre*incremento y *post*incremento.

Nos dejamos en el tintero unos pocos operadores («()», «[]», «->», «.», «,», y «*» unario. Los presentaremos cuando convenga y sepamos algo más de C.

> **C++**
>
> Ya debes entender de dónde viene el nombre C++: es un C «incrementado», o sea, mejorado. En realidad C++ es mucho más que un C con algunas mejoras: es un lenguaje orientado a objetos, así que facilita el diseño de programas siguiendo una filosofía diferente de la propia de los lenguajes imperativos y procedurales como C. Pero esa es otra historia.

En esta tabla te relacionamos todos los operadores (incluso los que aún no te hemos presentado con detalle) ordenados por precedencia (de mayor a menor) y con su aridad (número de operandos) y asociatividad:

Operador	Aridad	Asociatividad
() [] -> . $++_{\text{postfijo}}$ $--_{\text{postfijo}}$	2	izquierda
! ~ + - **sizeof** * & (*tipo*) $++_{\text{prefijo}}$ $--_{\text{prefijo}}$	1	derecha
* / %	2	izquierda
+ -	2	izquierda
<< >>	2	izquierda
< <= > >=	2	izquierda
== !=	2	izquierda
&	2	izquierda
^	2	izquierda
|	2	izquierda
&&	2	izquierda
||	2	izquierda
?:	3	izquierda
= += -= *= /= %= <<= >>= &= ^= |=	2	derecha
,	2	izquierda

EJERCICIOS

▶ 19 Sean a, b y c tres variables de tipo **int** cuyos valores actuales son 0, 1 y 2, respectivamente. ¿Qué valor tiene cada variable tras ejecutar esta secuencia de asignaciones?

```
a = b++ - c--;
a += --b;
c *= a + b;
a = b | c;
b = (a > 0) ? ++a : ++c;
b <<= a = 2;
c >>= a == 2;
a += a = b + c;
```

▶ 20 ¿Qué hace este programa?

ternario.c

```c
#include <stdio.h>

int main(void)
{
  int a, b, c, r;

  printf("Dame un valor entero: "); scanf("%d", &a);
  printf("Dame otro valor entero: "); scanf("%d", &b);
  printf("Y uno más: "); scanf("%d", &c);

  r = (a < b) ? ( (a < c) ? a : c ) : ( (b < c) ? b : c );

  printf("Resultado: %d\n", r);

  return 0;
}
```

▶ 21 Haz un programa que solicite el valor de x y muestre por pantalla el resultado de evaluar $x^4 - x^2 + 1$. (Recuerda que en C no hay operador de exponenciación.)

▶ 22 Diseña un programa C que solicite la longitud del lado de un cuadrado y muestre por pantalla su perímetro y su área.

▶ 23 Diseña un programa C que solicite la longitud de los dos lados de un rectángulo y muestre por pantalla su perímetro y su área.

▶ 24 Este programa C es problemático:

un_misterio.c

```c
#include <stdio.h>

int main(void)
{
  int a, b;

  a = 2147483647;
  b = a + a;
  printf("%d\n", a);
  printf("%d\n", b);
  return 0;
}
```

Al compilarlo y ejecutarlo hemos obtenido la siguiente salida por pantalla:

```
2147483647
-2
```

¿Qué ha ocurrido?

▶ 25 Diseña un programa C que solicite el radio r de una circunferencia y muestre por pantalla su perímetro ($2\pi r$) y su área (πr^2).

▶ 26 Si a es una variable de tipo **char** con el valor 127, ¿qué vale ~a? ¿Y qué vale !a? Y si a es una variable de tipo **unsigned int** con el valor 2147483647, ¿qué vale ~a? ¿Y qué vale !a?

▶ 27 ¿Qué resulta de evaluar cada una de estas dos expresiones?

a) `1 && !!!(0 || 1) || !(0 || 1)`

b) `1 & ~~~(0 | 1) | ~(0 | 1)`

▶ 28 ¿Por qué si a es una variable entera a / 2 proporciona el mismo resultado que a >> 1? ¿Con qué operación de bits puedes calcular a * 2? ¿Y a / 32? ¿Y a * 128?

▶ 29 ¿Qué hace este programa?

swap.c

```c
#include <stdio.h>

int main(void)
{
  unsigned char a, b;
  printf("Introduce el valor de a (entre 0 y 255): "); scanf("%hhu",&a);
  printf("Introduce el valor de b (entre 0 y 255): "); scanf("%hhu",&b);

  a ^= b;
  b ^= a;
  a ^= b;

  printf("Valor de a: %hhu\n", a);
  printf("Valor de b: %hhu\n", b);

  return 0;
}
```

(Nota: la forma en que hace lo que hace viene de un viejo truco de la programación en ensamblador, donde hay ricos juegos de instrucciones para la manipulación de datos bit a bit.)

1.13. Conversión implícita y explícita de tipos

El sistema de tipos escalares es más rígido que el de Python, aunque más rico. Cuando se evalúa una expresión y el resultado se asigna a una variable, has de tener en cuenta el tipo de todos los operandos y también el de la variable en la que se almacena.

Ilustraremos el comportamiento de C con fragmentos de programa que utilizan estas variables:

```c
char c;
int i;
float x;
```

¿5 > 3 > 2?

Recuerda que en Python podíamos combinar operadores de comparación para formar expresiones como $5 > 3 > 2$. Esa, en particular, se evalúa a *True*, pues 5 es mayor que 3 y 3 es mayor que 2. C también acepta esa expresión, pero con un significado completamente diferente basado en la asociatividad por la izquierda del operador >: en primer lugar evalúa la subexpresión $5 > 3$, que proporciona el valor «cierto»; pero como «cierto» es 1 (valor por defecto) y 1 no es mayor que 2, el resultado de la evaluación es 0, o sea, «falso».

¡Ojo con la interferencia entre ambos lenguajes! Problemas como éste surgirán con frecuencia cuando aprendas nuevos lenguajes: construcciones que significan algo en el lenguaje que conoces bien tienen un significado diferente en el nuevo.

Si asignas a un entero **int** el valor de un entero más corto, como un **char**, el entero corto *promociona* a un entero **int** automáticamente. Es decir, es posible efectuar esta asignación sin riesgo alguno:

```
i = c;
```

Podemos igualmente asignar un entero **int** a un **char**. C se encarga de hacer la conversión de tipos pertinente:

```
c = i;
```

Pero, ¿cómo? ¡En un byte (lo que ocupa un **char**) no caben cuatro (los que ocupa un **int**)! C toma los 8 bits menos significativos de *i* y los almacena en *c*, sin más. La conversión funciona correctamente, es decir, preserva el valor, sólo si el número almacenado en *i* está comprendido entre −128 y 127.

Observa este programa:

conversion_delicada.c

```
#include <stdio.h>

int main(void)
{
  int a, b;
  char c, d;

  a = 512;
  b = 127;
  c = a;
  d = b;
  printf("%hhd %hhd\n", c, d);

  return 0;
}
```

Produce esta salida por pantalla:

```
0 127
```

¿Por qué el primer resultado es 0? El valor 512, almacenado en una variable de tipo **int**, se representa con este patrón de bits: 00000000000000000000001000000000. Sus 8 bits menos significativos se almacenan en la variable *c* al ejecutar la asignación *c* = *a*, es decir, *c* almacena el patrón de bits 00000000, que es el valor decimal 0.

. . . . EJERCICIOS . . . .

▶ 30 ¿Qué mostrará por pantalla este programa?

otra_conversion_delicada.c otra_conversion_delicada.c

```
#include <stdio.h>

int main(void)
{
  int a, b;
  char c, d;
  unsigned char e, f;

  a = 384;
  b = 256;
  c = a;
  d = b;
  e = a;
  f = b;
  printf("%hhd␣%hhd\n", c, d);
  printf("%hhu␣%hhu\n", e, f);

  return 0;
}
```

Si asignamos un entero a una variable flotante, el entero promociona a su valor equivalente en coma flotante. Por ejemplo, esta asignación almacena en *x* el valor 2.0 (no el entero 2).

```
x = 2;
```

Si asignamos un valor flotante a un entero, el flotante se convierte en su equivalente entero (¡si lo hay!). Por ejemplo, la siguiente asignación almacena el valor 2 en *i* (no el flotante 2.0).

```
i = 2.0;
```

Y esta otra asignación almacena en *i* el valor 0:

```
i = 0.1;
```

EJERCICIOS

▶ 31 ¿Qué valor se almacena en las variables *i* (de tipo **int**) y *x* (de tipo **float**) tras ejecutar cada una de estas sentencias?

a) `i = 2;`
b) `i = 1 / 2;`
c) `i = 2 / 4;`
d) `i = 2.0 / 4;`
e) `x = 2.0 / 4.0;`
f) `x = 2.0 / 4;`
g) `x = 2 / 4;`
h) `x = 1 / 2;`

Aunque C se encarga de efectuar implícitamente muchas de las conversiones de tipo, puede que en ocasiones necesites indicar explícitamente una conversión de tipo. Para ello, debes preceder el valor a convertir con el tipo de destino encerrado entre paréntesis. Así:

```
i = (int) 2.3;
```

En este ejemplo da igual poner **(int)** que no ponerlo: C hubiera hecho la conversión implícitamente. El término **(int)** es el operador de conversión a enteros de tipo **int**. Hay un operador de conversión para cada tipo: **(char)**, **(unsigned int)** **(float)**, etc… Recuerda que el símbolo **(tipo)** es un operador unario conocido como *operador de coerción o conversión de tipos*.

EJERCICIOS

▶ 32 ¿Qué valor se almacena en las variables *i* (de tipo **int**) y *x* (de tipo **float**) tras ejecutar estas sentencias?

a) *i* = (**float**) 2;

b) *i* = 1 / (**float**) 2;

c) *i* = (**int**) (2 / 4);

d) *i* = (**int**) 2. / (**float**) 4;

e) *x* = 2.0 / (**int**) 4.0;

f) *x* = (**int**) 2.0 / 4;

g) *x* = (**int**) (2.0 / 4);

h) *x* = 2 / (**float**) 4;

i) *x* = (**float**) (1 / 2);

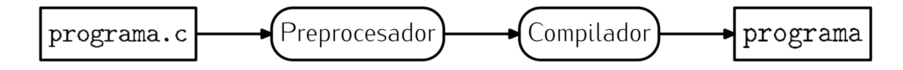
j) *x* = 1 / (**float**) 2;

. . . .

1.14. Las directivas y el preprocesador

Las líneas que empiezan con una palabra predecida por el carácter # son especiales. Las palabras que empiezan con # se denominan *directivas*. El compilador no llega a ver nunca las líneas que empiezan con una directiva. ¿Qué queremos decir exactamente con que no llega a verlas? El compilador gcc es, en realidad, un programa que controla varias etapas en el proceso de traducción de C a código de máquina. De momento, nos interesa considerar dos de ellas:

- el *preprocesador*,
- y el traductor de C a código de máquina (el *compilador* propiamente dicho).



El preprocesador es un programa independiente, aunque es infrecuente invocarlo directamente. El preprocesador del compilador gcc se llama cpp.

Las directivas son analizadas e interpretadas por el preprocesador. La directiva **#include** seguida del nombre de un fichero (entre los caracteres < y >) hace que el preprocesador sustituya la línea en la que aparece por el contenido íntegro del fichero (en inglés «include» significa «incluye»). El compilador, pues, no llega a ver la directiva, sino el resultado de su sustitución.

Nosotros sólo estudiaremos, de momento, dos directivas:

- **#define**, que permite definir constantes,
- e **#include**, que permite incluir el contenido de un fichero y que se usa para importar funciones, variables, constantes, etc. de bibliotecas.

1.15. Constantes

1.15.1. Definidas con la directiva *define*

Una diferencia de C con respecto a Python es la posibilidad que tiene el primero de definir constantes. Una constante es, en principio[15], una variable cuyo valor no puede ser modificado. Las constantes se definen con la directiva **#define**. Así:

```
#define CONSTANTE valor
```

Cada línea **#define** sólo puede contener el valor de una constante.

Por ejemplo, podemos definir los valores aproximados de π y del número *e* así:

```
#define PI 3.1415926535897931159979634685442
#define E  2.7182818284590450907955982984276
```

[15]Lo de «en principio» está justificado. No es cierto que las constantes de C sean variables. Lee el cuadro titulado «El preprocesador y las constantes» para saber qué son exactamente.

Intentar asignar un valor a PI o a E en el programa produce un error que detecta el compilador[16].

Observa que no hay operador de asignación entre el nombre de la constante y su valor y que la línea no acaba con punto y coma[17]. Es probable que cometas más de una vez el error de escribir el operador de asignación o el punto y coma.

No es obligatorio que el nombre de la constante se escriba en mayúsculas, pero sí un convenio ampliamente adoptado.

1.15.2. Definidas con el adjetivo const

C99 propone una forma alternativa de definir constantes mediante una nueva palabra reservada: **const**. Puedes usar **const** delante del tipo de una variable inicializada en la declaración para indicar que su valor no se modificará nunca.

constante.c

```
#include <stdio.h>

int main(void)
{
   const float pi = 3.14;
   float r, a;

   printf("Radio: ");
   scanf("%f", &r);

   a = pi * r * r;

   printf("Área: %f\n", a);

   return 0;
}
```

Pero la posibilidad de declarar constantes con **const** no nos libra de la directiva *define*, pues no son de aplicación en todo lugar donde conviene usar una constante. Más adelante, al estudiar la declaración de vectores, nos referiremos nuevamente a esta cuestión.

1.15.3. Con tipos enumerados

Es frecuente definir una serie de constantes con valores consecutivos. Imagina una aplicación en la que escogemos una opción de un menú como éste:

```
1) Cargar registros
2) Guardar registros
3) Añadir registro
4) Borrar registro
5) Modificar registro
6) Buscar registro
7) Finalizar
```

Cuando el usuario escoge una opción, la almacenamos en una variable (llamémosla *opcion*) y seleccionamos las sentencias a ejecutar con una serie de comparaciones como las que se muestran aquí esquemáticamente[18]:

[16]¿Has leído ya el cuadro «El preprocesador y las constantes»?

[17]¿A qué esperas para leer el cuadro «El preprocesador y las constantes»?

[18]Más adelante estudiaremos una estructura de selección que no es **if** y que se usa normalmente para especificar este tipo de acciones.

El preprocesador y las constantes

Como ya dijimos, el compilador no lee directamente nuestros ficheros con extensión «.c». Antes son tratados por otro programa al que se conoce como *preprocesador*. El preprocesador (que suele ser el programa *cpp*, por «C preprocessor») procesa las denominadas *directivas* (líneas que empiezan con #). Cuando el preprocesador encuentra la directiva **#define**, la elimina, pero recuerda la asociación establecida entre un identificador y un texto; cada vez que encuentra ese identificador en el programa, lo sustituye por el texto. Un ejemplo ayudará a entender el porqué de algunos errores misteriosos de C cuando se trabaja con constantes. Al compilar este programa:

preprocesar.c

```
#define PI 3.14

int main(void)
{
  int a = PI;
  return 0;
}
```

el preprocesador lo transforma en este otro programa (sin modificar nuestro fichero). Puedes comprobarlo invocando directamente al preprocesador:

```
$ cpp -P preprocesar.c ↲
```

El resultado es esto:

```
int main(void)
{
  int a = 3.14;
  return 0;
}
```

Como puedes ver, una vez «preprocesado», no queda ninguna directiva en el programa y la aparición del identificador PI ha sido sustituida por el texto 3.14. Un error típico es confundir un **#define** con una declaración normal de variables y, en consecuencia, poner una asignación entre el identificador y el valor:

```
#define PI = 3.14

int main(void)
{
  int a = PI;
  return 0;
}
```

El programa resultante es incorrecto. ¿Por qué? El compilador ve el siguiente programa tras ser preprocesado:

```
int main(void)
{
  int a = = 3.14;
  return 0;
}
```

¡La tercera línea del programa resultante no sigue la sintaxis del C!

```
if (opcion == 1) {
  Código para cargar registros
}
```

```
else if (opcion == 2) {
   Código para guardar registros
}
else if (opcion == 3) {
   ...
```

El código resulta un tanto ilegible porque no vemos la relación entre los valores numéricos y las opciones de menú. Es más legible recurrir a constantes:

```
#define CARGAR    1
#define GUARDAR   2
#define ANYADIR   3
#define BORRAR    4
#define MODIFICAR 5
#define BUSCAR    6
#define FINALIZAR 7

...

  if (opcion == CARGAR) {
     Código para cargar registros
  }
  else if (opcion == GUARDAR) {
     Código para guardar registros
  }
  else if (opcion == ANYADIR) {
     ...
```

Puedes ahorrarte la retahíla de **#define**s con los denominados tipos enumerados. Un tipo enumerado es un conjunto de valores «con nombre». Fíjate en este ejemplo:

```
enum { Cargar=1, Guardar, Anyadir, Borrar, Modificar, Buscar, Finalizar };

...

  if (opcion == Cargar) {
     Código para cargar registros
  }
  else if (opcion == Guardar) {
     Código para guardar registros
  }
  else if (opcion == Anyadir) {
     ...
```

La primera línea define los valores *Cargar*, *Guardar*, ... como una sucesión de valores correlativos. La asignación del valor 1 al primer elemento de la enumeración hace que la sucesión empiece en 1. Si no la hubiésemos escrito, la sucesión empezaría en 0.

Es habitual que los **enum** aparezcan al principio, tras los **#include** y **#define**.

. . . . EJERCICIOS . . . .

▶ 33 ¿Qué valor tiene cada identificador de este tipo enumerado?

```
enum { Primera='a', Segunda, Tercera, Penultima='y', Ultima };
```

(No te hemos explicado qué hace la segunda asignación. Comprueba que la explicación que das es correcta con un programa que muestre por pantalla el valor de cada identificador.)

. . . .

Los tipos enumerados sirven para algo más que asignar valores a opciones de menú. Es posible definir identificadores con diferentes valores para series de elementos como los días de la semana, los meses del año, etc.

```
enum { Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo };
enum { Invierno, Primavera, Verano, Otonyo };
enum { Rojo, Verde, Azul };
```

1.16. Las bibliotecas (módulos) se importan con **#include**

En C, los módulos reciben el nombre de *bibliotecas* (o *librerías*, como traducción fonéticamente similar del inglés *library*). La primera línea de `sumatorio.c` es ésta:

```
#include <stdio.h>
```

Con ella se indica que el programa hace uso de una biblioteca cuyas funciones, variables, tipos de datos y constantes están declaradas en el fichero `stdio.h`, que es abreviatura de «standard input/output» (entrada/salida estándar). En particular, el programa `sumatorio.c` usa las funciones *printf* y *scanf* de `stdio.h`. Los ficheros con extensión «`.h`» se denominan ficheros *cabecera* (la letra `h` es abreviatura de «header», que en inglés significa «cabecera»).

A diferencia de Python, C no permite importar un subconjunto de las funciones proporcionadas por una biblioteca. Al hacer **#include** de una cabecera se importan todas sus funciones, tipos de datos, variables y constantes. Es como si en Python ejecutaras la sentencia **from** *módulo* **import** *.

Normalmente no basta con incluir un fichero de cabecera con **#include** para poder compilar un programa que utiliza bibliotecas. Es necesario, además, compilar con opciones especiales. Abundaremos sobre esta cuestión inmediatamente, al presentar la librería matemática.

1.16.1. La biblioteca matemática

Podemos trabajar con funciones matemáticas incluyendo `math.h` en nuestros programas. La tabla 1.1 relaciona algunas de las funciones que ofrece la biblioteca matemática.

Todos los argumentos de las funciones de `math.h` son de tipo flotante.[19]

La biblioteca matemática también ofrece algunas constantes matemáticas predefinidas. Te relacionamos algunas en la tabla 1.2.

No basta con escribir **#include** `<math.h>` para poder usar las funciones matemáticas: has de compilar con la opción `-lm`:

```
$ gcc programa.c -lm -o programa ↲
```

¿Por qué? Cuando haces **#include**, el preprocesador introduce un fragmento de texto que dice qué funciones pasan a estar accesibles, pero ese texto no dice qué hace cada función y cómo lo hace (con qué instrucciones concretas). Si compilas sin `-lm`, el compilador se «quejará»:

```
$ gcc programa.c -o programa ↲
/tmp/ccm1nE0j.o: In function ‘main’:
/tmp/ccm1nE0j.o(.text+0x19): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

[19]Lo cierto es que son de tipo **double** (véase el apéndice A), pero no hay problema si las usas con valores y variables de tipo **float**, ya que hay conversión automática de tipos.

Función C	Función matemática
sqrt(*x*)	raíz cuadrada de *x*
sin(*x*)	seno de *x*
cos(*x*)	coseno de *x*
tan(*x*)	tangente de *x*
asin(*x*)	arcoseno de *x*
acos(*x*)	arcocoseno de *x*
atan(*x*)	arcotangente de *x*
exp(*x*)	el número *e* elevado a *x*
*exp*10(*x*)	10 elevado a *x*
log(*x*)	logaritmo en base *e* de *x*
*log*10(*x*)	logaritmo en base 10 de *x*
*log*2(*x*)	logaritmo en base 2 de *x*
pow(*x*, *y*)	*x* elevado a *y*
fabs(*x*)	valor absoluto de *x*
round(*x*)	redondeo al entero más próximo a *x*
ceil(*x*)	redondeo superior de *x*
floor(*x*)	redondeo inferior de *x*

Tabla 1.1: Algunas funciones matemáticas disponibles en la biblioteca `math.h`.

Constante	Valor
`M_E`	una aproximación del número e
`M_PI`	una aproximación del número π
`M_PI_2`	una aproximación de $\pi/2$
`M_PI_4`	una aproximación de $\pi/4$
`M_1_PI`	una aproximación de $1/\pi$
`M_SQRT2`	una aproximación de $\sqrt{2}$
`M_LOG2E`	una aproximación de $\log_2 e$
`M_LOG10E`	una aproximación de $\log_{10} e$

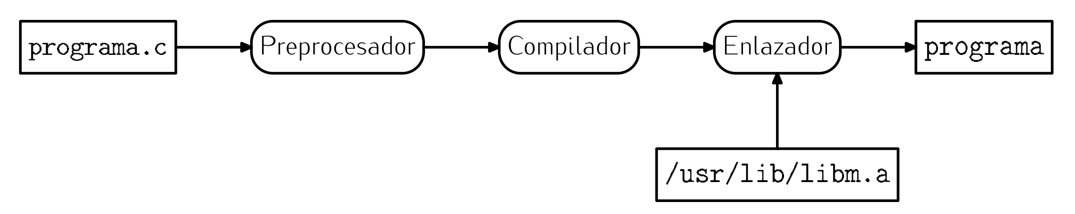
Tabla 1.2: Algunas constantes disponibles en la biblioteca `math.h`.

El mensaje advierte de que hay una «referencia indefinida a *sqrt*». En realidad no se está «quejando» el compilador, sino otro programa del que aún no te hemos dicho nada: el *enlazador* (en inglés, «linker»). El enlazador es un programa que detecta en un programa las llamadas a función no definidas en un programa C y localiza la definición de las funciones (ya compiladas) en bibliotecas. El fichero `math.h` que incluímos con **#define** contiene la cabecera de las funciones matemáticas, pero no su cuerpo. El cuerpo de dichas funciones, ya compilado (es decir, en código de máquina), reside en otro fichero: `/usr/lib/libm.a`. ¿Para qué vale el fichero `math.h` si no tiene el cuerpo de las funciones? Para que el compilador compruebe que estamos usando correctamente las funciones (que suministramos el número de argumentos adecuado, que su tipo es el que debe ser, etc.). Una vez que se comprueba que el programa es correcto, se procede a generar el código de máquina, y ahí es necesario «pegar» («enlazar») el código de máquina de las funciones matemáticas que hemos utilizado. El cuerpo ya compilado de *sqrt*, por ejemplo, se encuentra en `/usr/lib/libm.a` (`libm` es abreviatura de «math library»). El enlazador es el programa que «enlaza» el código de máquina de nuestro programa con el código de máquina de las bibliotecas que usamos. Con la opción `-lm` le indicamos al enlazador que debe resolver las referencias indefinidas a funciones matemáticas utilizando `/usr/lib/libm.a`.

La opción `-lm` evita tener que escribir `/usr/lib/libm.a` al final. Estas dos invocaciones del compilador son equivalentes:

```
$ gcc programa.c -o programa -lm ↲
$ gcc programa.c -o programa /usr/lib/libm.a ↲
```

El proceso completo de compilación cuando enlazamos con `/usr/lib/libm.a` puede representarse gráficamente así:



EJERCICIOS

▶ 34 Diseña un programa C que solicite la longitud de los tres lados de un triángulo (a, b y c) y muestre por pantalla su perímetro y su área ($\sqrt{s(s-a)(s-b)(s-c)}$, donde $s = (a + b + c)/2.$).

Compila y ejecuta el programa.

▶ 35 Diseña un programa C que solicite el radio r de una circunferencia y muestre por pantalla su perímetro ($2\pi r$) y su área (πr^2). Utiliza la aproximación a π predefinida en la biblioteca matemática.

Compila y ejecuta el programa.

1.17. Estructuras de control

Las estructuras de control de C son parecidas a las de Python. Bueno, hay alguna más y todas siguen unas reglas sintácticas diferentes. Empecemos estudiando las estructuras de control condicionales.

1.17.1. Estructuras de control condicionales

La sentencia de selección if

La estructura de control condicional fundamental es el **if**. En C se escribe así:

```
if (condición) {
  sentencias
}
```

Los paréntesis que encierran a la condición *son obligatorios*. Como en Python no lo son, es fácil que te equivoques por no ponerlos. Si el bloque de sentencias consta de una sola sentencia, no es necesario encerrarla entre llaves:

```
if (condición)
  sentencia;
```

La sentencia de selección if-else

Hay una forma **if-else**, como en Python:

```
if (condición) {
  sentencias_si
}
else {
  sentencias_no
}
```

Si uno de los bloques sólo tiene una sentencia, generalmente puedes eliminar las llaves:

```
if (condición)
  sentencia_si;
else {
  sentencias_no
}

if (condición) {
  sentencias_si
}
else
  sentencia_no;

if (condición)
  sentencia_si;
else
  sentencia_no;
```

Ojo: la indentación no significa nada para el compilador. La ponemos únicamente para facilitar la lectura. Pero si la indentación no significa nada nos enfrentamos a un problema de ambigüedad con los **if** anidados:

```
if (condición)
  if (otra_condición) {
    sentencias_si
  }
else { // ¿¿¿???
  sentencias_no
}
```

¿A cuál de los dos **if** pertenece el **else**? ¿Hará el compilador de C una interpretación como la que sugiere la indentación en el último fragmento o como la que sugiere este otro?:

```
if (condición)
  if (otra_condición) {
    sentencias_si
  }
  else { // ¿¿¿???
    sentencias_no
  }
```

C rompe la ambigüedad trabajando con esta sencilla regla: *el* **else** *pertenece al* **if** *«libre» más cercano*. Si quisiéramos expresar la primera estructura, deberíamos añadir llaves para determinar completamente qué bloque está dentro de qué otro:

```
if (condición) {
  if (otra_condición) {
    sentencias_si
```

```
    }
  }
  else {
    sentencias_no
  }
```

A pesar de que el **if** externo contiene una sola sentencia (otro y, por tanto, las llaves son redundantes**if**), las llaves son necesarias para indicar que el **else** va asociado a la condición exterior.

No hay sentencia elif: la combinación else if

C no tiene una estructura *elif* como la de Python, pero tampoco la necesita. Puedes usar **else if** donde hubieras puesto un **elif** en Python:

```
if (condición) {
  sentencias_si
}
else if (condición2) {
  sentencias_si2
}
else if (condición3) {
  sentencias_si3
}
else {
  sentencias_no
}
```

. . . . EJERCICIOS . . . .

▶ 36 Diseña un programa C que pida por teclado un número entero y diga si es par o impar.

▶ 37 Diseña un programa que lea dos números enteros y muestre por pantalla, de estos tres mensajes, el que convenga:

- «`El segundo es el cuadrado exacto del primero.`»,
- «`El segundo es menor que el cuadrado del primero.`»,
- «`El segundo es mayor que el cuadrado del primero.`».

▶ 38 También en C es problemática la división por 0. Haz un programa C que resuelva la ecuación $ax + b = 0$ solicitando por teclado el valor de a y b (ambos de tipo **float**). El programa detectará si la ecuación no tiene solución o si tiene infinitas soluciones y, en cualquiera de los dos casos, mostrará el pertinente aviso.

▶ 39 Diseña un programa que solucione ecuaciones de segundo grado. El programa detectará y tratará por separado las siguientes situaciones:

- la ecuación tiene dos soluciones reales;
- la ecuación tiene una única solución real;
- la ecuación no tiene solución real;
- la ecuación tiene infinitas soluciones.

▶ 40 Realiza un programa que proporcione el desglose en billetes y monedas de una cantidad exacta de euros. Hay billetes de 500, 200, 100, 50, 20, 10 y 5 euros y monedas de 1 y 2 euros.

Por ejemplo, si deseamos conocer el desglose de 434 euros, el programa mostrará por pantalla el siguiente resultado:

```
2 billetes de 200 euros.
1 billete de 20 euros.
1 billete de 10 euros.
2 monedas de 2 euros.
```

Observa que la palabra «billete» (y «moneda») concuerda en número con la cantidad de billetes (o monedas) y que si no hay piezas de un determinado tipo (en el ejemplo, de 1 euro), no muestra el mensaje correspondiente.

▶ 41 Diseña un programa C que lea un carácter cualquiera desde el teclado, y muestre el mensaje «`Es una MAYÚSCULA.`» cuando el carácter sea una letra mayúscula y el mensaje «`Es una MINÚSCULA.`» cuando sea una minúscula. En cualquier otro caso, no mostrará mensaje alguno. (Considera únicamente letras del alfabeto inglés.)

▶ 42 Diseña un programa que lea cinco números enteros por teclado y determine cuál de los cuatro últimos números es más cercano al primero.

(Por ejemplo, si el usuario introduce los números 2, 6, 4, 1 y 10, el programa responderá que el número más cercano al 2 es el 1.)

▶ 43 Diseña un programa que, dado un número entero, determine si éste es el doble de un número impar.

(Ejemplo: 14 es el doble de 7, que es impar.)

La sentencia de selección switch

Hay una estructura condicional que no existe en Python: la estructura de selección múltiple. Esta estructura permite seleccionar un bloque de sentencias en función del valor de una expresión (típicamente una variable).

```
switch (expresión) {
  case valor1:
    sentencias
    break;
  case valor2:
    sentencias
    break;
  ...
  default:
    sentencias
    break;
}
```

El fragmento etiquetado con **default** es opcional.

Para ilustrar el uso de **switch**, nada mejor que un programa que muestra algo por pantalla en función de la opción seleccionada de un menú:

menu.c

```
#include <stdio.h>

int main(void)
{
  int opcion;

  printf("1) Saluda\n");
  printf("2) Despídete\n");
  scanf("%d", &opcion);
```

```c
    switch (opcion) {
      case 1:
        printf("Hola\n");
        break;
      case 2:
        printf("Adiós\n");
        break;
      default:
        printf("Opción no válida\n");
        break;
    }
    return 0;
}
```

Aunque resulta algo más elegante esta otra versión, que hace uso de tipos enumerados:

menu_1.c   menu.c

```c
#include <stdio.h>

enum { Saludar=1, Despedirse };

int main(void)
{
    int opcion;

    printf("1) Saluda\n");
    printf("2) Despídete\n");
    scanf("%d", &opcion);
    switch (opcion) {
      case Saludar:
        printf("Hola\n");
        break;
      case Despedirse:
        printf("Adiós\n");
        break;
      default:
        printf("Opción no válida\n");
        break;
    }
    return 0;
}
```

Un error típico al usar la estructura **switch** es olvidar el **break** que hay al final de cada opción. Este programa, por ejemplo, presenta un comportamiento curioso:

menu2.c   menu2.c

```c
#include <stdio.h>

enum { Saludar=1, Despedirse };

int main(void)
{
    int opcion;

    printf("1) Saluda\n");
    printf("2) Despídete\n");
    scanf("%d", &opcion);
    switch (opcion) {
      case Saludar:
```

```
      printf("Hola\n");
    case Despedirse:
      printf("Adiós\n");
    default:
      printf("Opción␣no␣válida\n");
  }
  return 0;
}
```

Si seleccionas la opción 1, no sale un único mensaje por pantalla, ¡salen tres: `Hola`, `Adiós` y `Opción no válida`! Y si seleccionas la opción 2, ¡salen dos mensajes: `Adiós` y `Opción no válida`! Si no hay **break**, el flujo de control que entra en un **case** ejecuta las acciones asociadas al siguiente **case**, y así hasta encontrar un **break** o salir del **switch** por la última de sus líneas.

El compilador de C no señala la ausencia de **break** como un error porque, de hecho, no lo es. Hay casos en los que puedes explotar a tu favor este curioso comportamiento del **switch**:

menu3.c

```
#include <stdio.h>

enum { Saludar=1, Despedirse, Hola, Adios };

int main(void)
{
  int opcion;

  printf("1)␣Saluda\n");
  printf("2)␣Despídete\n");
  printf("3)␣Di␣hola\n");
  printf("4)␣Di␣adiós\n");
  scanf("%d", &opcion);
  switch (opcion) {
    case Saludar:
    case Hola:
      printf("Hola\n");
      break;
    case Despedirse:
    case Adios:
      printf("Adiós\n");
      break;
    default:
      printf("Opción␣no␣válida\n");
      break;
  }
  return 0;
}
```

¿Ves por qué?

1.17.2. Estructuras de control iterativas

El bucle while

El bucle **while** de Python se traduce casi directamente a C:

```
while (condición) {
  sentencias
}
```

Nuevamente, los paréntesis son obligatorios y las llaves pueden suprimirse si el bloque contiene una sola sentencia.

Veamos un ejemplo de uso: un programa que calcula x^n para x y n enteros:

potencia.c

```
#include <stdio.h>

int main(void)
{
  int x, n, i, r;

  printf("x: "); scanf("%d", &x);
  printf("n: "); scanf("%d", &n);
  r = 1;
  i = 0;
  while (i < n) {
    r *= x;
    i++;
  }
  printf("%d**%d = %d\n", x, n, r);

  return 0;
}
```

El bucle do-while

Hay un bucle iterativo que Python no tiene: el **do-while**:

```
do {
  sentencias
} while (condición);
```

El bucle **do-while** evalúa la condición tras cada ejecución de su bloque, así que es seguro que éste se ejecuta al menos una vez. Podríamos reescribir `sumatorio.c` para usar un bucle **do-while**:

sumatorio.c

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  int a, b, i;
  float s;

  /* Pedir límites inferior y superior. */
  do {
    printf("Límite inferior:"); scanf("%d", &a);
    if (a < 0) printf("No puede ser negativo\n");
  } while (a < 0);

  do {
    printf("Límite superior:"); scanf("%d", &b);
    if (b < a) printf("No puede ser menor que %d\n", a);
  } while (b < a);

  /* Calcular el sumatorio de la raíz cuadrada de i para i entre a y b. */
  s = 0.0;
  for (i = a; i <= b; i++) s += sqrt(i);

```

```
    /* Mostrar el resultado. */
    printf("Sumatorio␣de␣raíces␣de␣%d␣a␣%d:␣%f\n", a, b, s);

    return 0;
}
```

Los bucles **do-while** no añaden potencia al lenguaje, pero sí lo dotan de mayor expresividad. Cualquier cosa que puedas hacer con bucles **do-while**, puedes hacerla también con sólo bucles **while** y la ayuda de alguna sentencia condicional **if**, pero probablemente requerirán mayor esfuerzo por tu parte.

. . . . EJERCICIOS . . . .

▶ 44 Escribe un programa que muestre un menú en pantalla con dos opciones: «saludar» y «salir». El programa pedirá al usuario una opción y, si es válida, ejecutará su acción asociada. Mientras no se seleccione la opción «salir», el menú reaparecerá y se solicitará nuevamente una opción. Implementa el programa haciendo uso únicamente de bucles **do-while**.

▶ 45 Haz un programa que pida un número entero de teclado distinto de 1. A continuación, el programa generará una secuencia de números enteros cuyo primer número es el que hemos leído y que sigue estas reglas:

- si el último número es par, el siguiente resulta de dividir a éste por la mitad;
- si el último número es impar, el siguiente resulta de multiplicarlo por 3 y añadirle 1.

Todos los números se irán mostrando por pantalla conforme se vayan generando. El proceso se repetirá hasta que el número generado sea igual a 1. Utiliza un bucle **do-while**.

. . . .

El bucle for

El bucle **for** de Python existe en C, pero con importantes diferencias.

```
for (inicialización; condición; incremento) {
    sentencias
}
```

Los paréntesis de la primera línea son obligatorios. Fíjate, además, en que los tres elementos entre paréntesis se separan con puntos y comas.

El bucle **for** presenta tres componentes. Es equivalente a este fragmento de código:

```
inicialización;
while (condición) {
    sentencias
    incremento;
}
```

Una forma habitual de utilizar el bucle **for** es la que se muestra en este ejemplo, que imprime por pantalla los números del 0 al 9 y en el que suponemos que *i* es de tipo **int**:

```
for (i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

Es equivalente, como decíamos, a este otro fragmento de programa:

Comparaciones y asignaciones

Un error frecuente es sustituir el operador de comparación de igualdad por el de asignación en una estructura **if** o **while**. Analiza este par de sentencias:

```
a = 0
if (a = 0) { // Lo que escribió... ¿bien o mal?
  ...
}
```

Parece que la condición del **if** se evalúa a cierto, pero no es así: la «comparación» es, en realidad, una asignación. El resultado es que *a* recibe el valor 0 y que ese 0, devuelto por el operador de asignación, se considera la representación del valor «falso». Lo correcto hubiera sido:

```
a = 0
if (a == 0) { // Lo que quería escribir.
  ...
}
```

Aunque esta construcción es perfectamente válida, provoca la emisión de un mensaje de error en muchos compiladores, pues suele ser fruto de un error.

Los programadores más disciplinados evitan cometer este error escribiendo siempre la variable en la parte derecha:

```
a = 0
if (0 == a) { // Correcto.
  ...
}
```

De ese modo, si se confunden y usan = en lugar de ==, se habrá escrito una expresión incorrecta y el compilador detendrá el proceso de traducción a código de máquina:

```
a = 0
if (0 = a) { // Mal: error detectable por el compilador.
  ...
}
```

```
i = 0;
while (i < 10) {
  printf("%d\n", i);
  i++;
}
```

. . . . EJERCICIOS . . . .

▶ **46** Implementa el programa de cálculo de x^n (para x y n entero) con un bucle **for**.

▶ **47** Implementa un programa que dado un número de tipo **int**, leído por teclado, se asegure de que sólo contiene ceros y unos y muestre su valor en pantalla si lo interpretamos como un número binario. Si el usuario introduce, por ejemplo, el número 1101, el programa mostrará el valor 13. Caso de que el usuario introduzca un número formado por números de valor diferente, indica al usuario que no puedes proporcionar el valor de su interpretación como número binario.

▶ **48** Haz un programa que solicite un número entero y muestre su factorial. Utiliza un entero de tipo **long long** para el resultado. Debes usar un bucle **for**.

▶ **49** El número de combinaciones de n elementos tomados de m en m es:

$$C_n^m = \binom{n}{m} = \frac{n!}{(n-m)!\,m!}.$$

Diseña un programa que pida el valor de n y m y calcule C_n^m. (Ten en cuenta que n ha de ser mayor o igual que m.)

(Puedes comprobar la validez de tu programa introduciendo los valores $n = 15$ y $m = 10$: el resultado es 3003.)

▶ **50** ¿Qué muestra por pantalla este programa?

desplazamientos.c

```c
#include <stdio.h>

int main(void)
{
   int a = 127, b = 1024, c, i;

   c = a ^ b;

   printf("%d\n", c);

   a = 2147483647;
   for (i = 0; i < 8*sizeof(a); i++) {
      printf("%d", ((c & a) != 0) ? 1 : 0);
      a >>= 1;
   }
   printf("\n");

   a = 1;
   for (i = 0; i < 8*sizeof(a); i++) {
      if ((c & a) != 0) c >>= 1;
      else c <<= 1;
      a <<= 1;
   }

   a = 2147483647;
   for (i = 0; i < 8*sizeof(a); i++) {
      printf("%d", ((c & a) != 0) ? 1 : 0);
      a >>= 1;
   }
   printf("\n");
   return 0;
}
```

▶ **51** Cuando no era corriente el uso de terminales gráficos de alta resolución era común representar gráficas de funciones con el terminal de caracteres. Por ejemplo, un periodo de la función seno tiene este aspecto al representarse en un terminal de caracteres (cada punto es un asterisco):

```
                        *
                                 *
                                           *
                                                   *
                                                         *
                                                              *
                                                               *
                                                               *
                                                            *
                                                      *
                                               *
                                      *
                             *
                  *
         *
*
```

```
       *
   *
 *
 *
  *
     *
         *
              *
```

Haz un programa C que muestre la función seno utilizando un bucle que recorre el periodo 2π en 24 pasos (es decir, representándolo con 24 líneas).

▶ 52 Modifica el programa para que muestre las funciones seno (con asteriscos) y coseno (con sumas) simultáneamente.

. . . .

Variables de bucle de usar y tirar

C99 ha copiado una buena idea de C++: permitir que las variables de bucle se definan allí donde se usan y dejen de existir cuando el bucle termina. Fíjate en este programa:

for_con_variable.c — for_con_variable.c

```
#include <stdio.h>

int main(void)
{
  int a = 1;

  for (int i = 0; i < 32; i++) {
    printf("2**%2d␣=␣%10u\n", i, a);
    a <<= 1;
  }

  return 0;
}
```

La variable *i*, el índice del bucle, se declara en la mismísima zona de inicialización del bucle. La variable *i* sólo existe en el ámbito del bucle, que es donde se usa.

Hacer un bucle que recorra, por ejemplo, los números pares entre 0 y 10 es sencillo: basta sustituir el modo en que se incrementa la variable índice:

```
for (i = 0; i < 10; i = i + 2) {
  printf("%d\n", i);
}
```

aunque la forma habitual de expresar el incremento de *i* es esta otra:

```
for (i = 0; i < 10; i += 2) {
  printf("%d\n", i);
}
```

Un bucle que vaya de 10 a 1 en orden inverso presenta este aspecto:

```
for (i = 10; i > 0; i--) {
  printf("%d\n", i);
}
```

. . . . EJERCICIOS . . . .

▶ 53 Diseña un programa C que muestre el valor de 2^n para todo n entre 0 y un valor entero proporcionado por teclado.

▶ 54 Haz un programa que pida al usuario una cantidad de euros, una tasa de interés y un número de años y muestre por pantalla en cuánto se habrá convertido el capital inicial transcurridos esos años si cada año se aplica la tasa de interés introducida.

Recuerda que un capital C a un interés del x por cien durante n años se convierte en $C \cdot (1 + x/100)^n$.

(Prueba tu programa sabiendo que 10 000 euros al 4.5% de interés anual se convierten en 24 117.14 euros al cabo de 20 años.)

▶ 55 Un vector en un espacio tridimensional es una tripleta de valores reales (x, y, z). Deseamos confeccionar un programa que permita operar con dos vectores. El usuario verá en pantalla un menú con las siguientes opciones:

```
1) Introducir el primer vector
2) Introducir el segundo vector
3) Calcular la suma
4) Calcular la diferencia
5) Calcular el producto vectorial
6) Calcular el producto escalar
7) Calcular el ángulo (en grados) entre ellos
8) Calcular la longitud
9) Finalizar
```

Tras la ejecución de cada una de las acciones del menú éste reaparecerá en pantalla, a menos que la opción escogida sea la número 9. Si el usuario escoge una opción diferente, el programa advertirá al usuario de su error y el menú reaparecerá.

Las opciones 4 y 5 pueden proporcionar resultados distintos en función del orden de los operandos, así que, si se escoge cualquiera de ellas, aparecerá un nuevo menú que permita seleccionar el orden de los operandos. Por ejemplo, la opción 4 mostrará el siguiente menú:

```
1) Primer vector menos segundo vector
2) Segundo vector menos primer vector
```

Nuevamente, si el usuario se equivoca, se le advertirá del error y se le permitirá corregirlo.

La opción 8 del menú principal conducirá también a un submenú para que el usuario decida sobre qué vector se aplica el cálculo de longitud.

Puede que necesites que te refresquemos la memoria sobre los cálculos a realizar. Quizá la siguiente tabla te sea de ayuda:

Operación	**Cálculo**
Suma: $(x_1, y_1, z_1) + (x_2, y_2, z_2)$	$(x_1 + x_2, y_1 + y_2, z_1 + z_2)$
Diferencia: $(x_1, y_1, z_1) - (x_2, y_2, z_2)$	$(x_1 - x_2, y_1 - y_2, z_1 - z_2)$
Producto escalar: $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2)$	$x_1x_2 + y_1y_2 + z_1z_2$
Producto vectorial: $(x_1, y_1, z_1) \times (x_2, y_2, z_2)$	$(y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$
Ángulo entre (x_1, y_1, z_1) y (x_2, y_2, z_2)	$\frac{180}{\pi} \cdot \arccos\left(\frac{x_1x_2 + y_1y_2 + z_1z_2}{\sqrt{x_1^2 + y_1^2 + z_1^2}\sqrt{x_2^2 + y_2^2 + z_2^2}}\right)$
Longitud de (x, y, z)	$\sqrt{x^2 + y^2 + z^2}$

Ten en cuenta que tu programa debe contemplar toda posible situación excepcional: divisiones por cero, raíces con argumento negativo, etc..

1.17.3. Sentencias para alterar el flujo iterativo

La sentencia **break** también está disponible en C. De hecho, ya hemos visto una aplicación suya en la estructura de control **switch**. Con ella puedes, además, abortar al instante la ejecución de un bucle cualquiera (**while**, **do-while** o **for**).

Otra sentencia de C que puede resultar útil es **continue**. Esta sentencia finaliza la iteración *actual*, pero no aborta la ejecución del bucle.

Por ejemplo, cuando en un bucle **while** se ejecuta **continue**, la siguiente sentencia a ejecutar es la condición del bucle; si ésta se cumple, se ejecutará una nueva iteración del bucle.

. . . . EJERCICIOS . . . .

▶ 56 ¿Qué muestra por pantalla este programa?

continue.c

```
#include <stdio.h>

int main(void)
{
  int i;

  i = 0;
  while (i < 10) {
    if (i % 2 == 0) {
      i++;
      continue;
    }
    printf("%d\n", i);
    i++;
  }

  for (i = 0; i < 10; i++) {
    if (i % 2 != 0)
      continue;
    printf("%d\n", i);
  }
  return 0;
}
```

▶ 57 Traduce a C este programa Python.

```
car = raw_input('Dame␣un␣carácter:␣')
if "a" <= car.lower() <= "z" or car == "_":
  print "Este␣carácter␣es␣válido␣en␣un␣identificador␣en␣Python."
else:
  if not (car < "0" or "9" < car):
    print "Un␣dígito␣es␣válido␣en␣un␣identificador␣en␣Python,",
    print "siempre␣que␣no␣sea␣el␣primer␣carácter."
  else:
    print "Carácter␣no␣válido␣para␣formar␣un␣identificador␣en␣Python."
```

▶ 58 Traduce a C este programa Python.

```
from math import pi
radio = float(raw_input('Dame␣el␣radio␣de␣un␣círculo:␣'))
opcion = ''
while opcion != 'a' and opcion != 'b' and opcion != 'c':
  print 'Escoge␣una␣opción:␣'
  print 'a)␣Calcular␣el␣diámetro.'
  print 'b)␣Calcular␣el␣perímetro.'
```

```
    print 'c) Calcular el área.'
    opcion = raw_input('Teclea a, b o c y pulsa el retorno de carro: ')
    if opcion == 'a':
        diametro = 2 * radio
        print 'El diámetro es', diametro
    elif opcion == 'b':
        perimetro = 2 * pi * radio
        print 'El perímetro es', perimetro
    elif opcion == 'c':
        area = pi * radio ** 2
        print 'El área es', area
    else:
        print 'Sólo hay tres opciones: a, b o c. Tú has tecleado', opcion
```

▶ 59 Traduce a C este programa Python.

```
anyo = int(raw_input('Dame un año: '))
if anyo % 4 == 0 and (anyo % 100 != 0 or anyo % 400 == 0):
    print 'El año', anyo, 'es bisiesto.'
else:
    print 'El año', anyo, 'no es bisiesto.'
```

▶ 60 Traduce a C este programa Python.

```
limite = int(raw_input('Dame un número: '))

for num in range(1, limite+1):
    creo_que_es_primo = 1
    for divisor in range(2, num):
        if num % divisor == 0:
            creo_que_es_primo = 0
            break
    if creo_que_es_primo == 1:
        print num
```

▶ 61 Escribe un programa que solicite dos enteros n y m asegurándose de que m sea mayor o igual que n. A continuación, muestra por pantalla el valor de $\sum_{i=n}^{m} 1/i$.

▶ 62 Escribe un programa que solicite un número entero y muestre todos los números primos entre 1 y dicho número.

▶ 63 Haz un programa que calcule el máximo común divisor (mcd) de dos enteros positivos. El mcd es el número más grande que divide exactamente a ambos números.

▶ 64 Haz un programa que calcule el máximo común divisor (mcd) de tres enteros positivos.

▶ 65 Haz un programa que vaya leyendo números y mostrándolos por pantalla hasta que el usuario introduzca un número negativo. En ese momento, el programa acabará mostrando un mensaje de despedida.

▶ 66 Haz un programa que vaya leyendo números hasta que el usuario introduzca un número negativo. En ese momento, el programa mostrará por pantalla el número mayor de cuantos ha visto.

Capítulo 2

Estructuras de datos en C: vectores estáticos y registros

> *—Me llamo Alicia, Majestad —dijo Alicia con mucha educación; pero añadió para sus adentros: «¡Vaya!, en realidad no son más que un mazo de cartas. ¡No tengo por qué tenerles miedo!».*
>
> LEWIS CARROLL, *Alicia en el País de las Maravillas*.

En este capítulo vamos a estudiar algunas estructuras que agrupan varios datos, pero cuyo tamaño resulta conocido al compilar el programa y no sufre modificación alguna durante su ejecución. Empezaremos estudiando los *vectores*, estructuras que se pueden asimilar a las listas Python. En C, las *cadenas* son un tipo particular de vector. Manejar cadenas en C resulta más complejo y delicado que manejarlas en Python. Como contrapartida, es más fácil definir en C *vectores multidimensionales* (como las matrices) que en Python. En este capítulo nos ocuparemos también de ellos. Estudiaremos además los *registros* en C, que permiten definir nuevos tipos como agrupaciones de datos de tipos no necesariamente idénticos. Los registros de C son conceptualmente idénticos a los que estudiamos en Python.

2.1. Vectores estáticos

Un vector (en inglés, «array») es una secuencia de valores a los que podemos acceder mediante índices que indican sus respectivas posiciones. Los vectores pueden asimilarse a las listas Python, pero con una limitación fundamental: *todos los elementos del vector han de tener el mismo tipo.* Podemos definir vectores de enteros, vectores de flotantes, etc., pero no podemos definir vectores que, por ejemplo, contengan a la vez enteros y flotantes. El tipo de los elementos de un vector se indica en la declaración del vector.

C nos permite trabajar con vectores estáticos y dinámicos. En este capítulo nos ocupamos únicamente de los denominados vectores estáticos, que son aquellos que tienen *tamaño fijo y conocido en tiempo de compilación*. Es decir, el número de elementos del vector no puede depender de datos que suministra el usuario: se debe hacer explícito mediante una expresión que podamos evaluar examinando únicamente el texto del programa.

2.1.1. Declaración de vectores

Un vector *a* de 10 enteros de tipo **int** se declara así:

```
int a[10];
```

> ***Sin cortes***
>
> Los vectores C son mucho más limitados que las listas Python. A los problemas relacionados con el tamaño fijo de los vectores o la homogeneidad en el tipo de sus elementos se une una incomodidad derivada de la falta de operadores a los que nos hemos acostumbrado como programadores Python. El operador de corte, por ejemplo, no existe en C. Cuando en Python deseábamos extraer una copia de los elementos entre *i* y *j* de un vector *a* escribíamos `a[i:j+1]`. En C no hay operador de corte... ni operador de concatenación o repetición, ni sentencias de borrado de elementos, ni se entienden como accesos desde el final los índices negativos, ni hay operador de pertenencia, etc. Echaremos de menos muchas de las facilidades propias de Python.

El vector *a* comprende los elementos `a[0]`, `a[1]`, `a[2]`, ..., `a[9]`, todos de tipo **int**. Al igual que con las listas Python, los índices de los vectores C empiezan en cero.

En una misma línea puedes declarar más de un vector, siempre que todos compartan el mismo tipo de datos para sus componentes. Por ejemplo, en esta línea se declaran dos vectores de **float**, uno con 20 componentes y otro con 100:

```
float a[20], b[100];
```

También es posible mezclar declaraciones de vectores y escalares en una misma línea. En este ejemplo se declaran las variables *a* y *c* como vectores de 80 caracteres y la variable *b* como escalar de tipo carácter:

```
char a[80], b, c[80];
```

Se considera mal estilo declarar la talla de los vectores con literales de entero. Es preferible utilizar algún identificador para la talla, pero teniendo en cuenta que éste debe corresponder a una constante:

```
#define TALLA 80
...
char a[TALLA];
```

Esta otra declaración es incorrecta, pues usa una variable para definir la talla del vector[1]:

```
int talla = 80;
...
char a[talla]; // ¡No siempre es válido!
```

Puede que consideres válida esta otra declaración que prescinde de constantes definidas con *define* y usa constantes declaradas con **const**, pero no es así:

```
const int talla = 80;
...
char a[talla]; // ¡No siempre es válido!
```

Una variable **const** es una variable en toda regla, aunque de «sólo lectura».

2.1.2. Inicialización de los vectores

Una vez creado un vector, sus elementos presentan valores arbitrarios. *Es un error suponer que los valores del vector son nulos tras su creación.* Si no lo crees, fíjate en este programa:

[1] Como siempre, hay excepciones: C99 permite declarar la talla de un vector con una expresión cuyo valor sólo se conoce en tiempo de ejecución, pero sólo si el vector es una variable local a una función. Para evitar confusiones, no haremos uso de esa característica en este capítulo y lo consideraremos incorrecto.

sin_inicializar.c

```
#include <stdio.h>

#define TALLA 5

int main(void)
{
  int i, a[TALLA];

  for (i = 0; i < TALLA; i++)
    printf("%d\n", a[i]);
  return 0;
}
```

Observa que el acceso a elementos del vector sigue la misma notación de Python: usamos el identificador del vector seguido del índice encerrado entre corchetes. En una ejecución del programa obtuvimos este resultado en pantalla (es probable que obtengas resultados diferentes si repites el experimento):

```
1073909760
1075061012
1205
1074091790
1073941880
```

Evidentemente, no son cinco ceros.

Podemos inicializar todos los valores de un vector a cero con un bucle **for**:

inicializados_a_cero.c

```
#include <stdio.h>

#define TALLA 10

int main(void)
{
  int i, a[TALLA];

  for (i = 0; i < TALLA; i++)
    a[i] = 0;

  for (i = 0; i < TALLA; i++)
    printf("%d\n", a[i]);

  return 0;
}
```

EJERCICIOS

▶ 67 Declara e inicializa un vector de 100 elementos de modo que los componentes de índice par valgan 0 y los de índice impar valgan 1.

▶ 68 Escribe un programa C que almacene en un vector los 50 primeros números de Fibonacci. Una vez calculados, el programa los mostrará por pantalla en orden inverso.

▶ 69 Escribe un programa C que almacene en un vector los 50 primeros números de Fibonacci. Una vez calculados, el programa pedirá al usuario que introduzca un número y dirá si es o no es uno de los 50 primeros números de Fibonacci.

Cuestión de estilo: ¿constantes o literales al declarar la talla de un vector?

¿Por qué es preferible declarar el tamaño de un vector con una constante? Porque la talla del vector puede aparecer en diferentes puntos del programa y es posible que algún día modifiquemos el programa para trabajar con una talla diferente. En tal caso, deberíamos editar muchas líneas diferentes del programa (quizá decenas o cientos). Bastaría que olvidásemos modificar una o que modificásemos una de más para que el programa fuera erróneo. Fíjate en este programa C:

```
#include <stdio.h>

int main(void)
{
  int i, a[10], b[10];

  for (i = 0; i < 10; i++) a[i] = 0;
  for (i = 0; i < 10; i++) b[i] = 0;
  for (i = 0; i < 10; i++) printf("%d\n", a[i]);
  for (i = 0; i < 10; i++) printf("%d\n", b[i]);
  return 0;
}
```

Las tallas de *a* y *b* aparecen en seis lugares. Imagina que deseas modificar el programa para que *a* pase a tener 20 enteros: tendrás que modificar sólo tres de esos dieces. Ello te obliga a leer el programa detenidamente y, cada vez que encuentres un 10, pensar si ese 10 en particular es o no es la talla de *a*. Complicado. Estudia esta versión:

```
1  #include <stdio.h>
2  
3  #define TALLA_A 10
4  #define TALLA_B 10
5  
6  int main(void)
7  {
8    int i, a[TALLA_A], b[TALLA_B];
9  
10   for (i = 0; i < TALLA_A; i++) a[i] = 0;
11   for (i = 0; i < TALLA_B; i++) b[i] = 0;
12   for (i = 0; i < TALLA_A; i++) printf("%d\n", a[i]);
13   for (i = 0; i < TALLA_B; i++) printf("%d\n", b[i]);
14   return 0;
15 }
```

Si ahora has de modificar *a* para que tenga 20 elementos, basta con editar la línea 3: cambia el 10 por un 20. Más rápido y con mayor garantía de no cometer errores. ¿Por qué en Python no nos preocupó esta cuestión? Recuerda que en Python no había declaración de variables, que las listas podían modificar su longitud durante la ejecución de los programas y que podías consultar la longitud de cualquier secuencia de valores con la función predefinida *len*. Python ofrece mayores facilidades al programador, pero a un precio: menos velocidad de ejecución y más consumo de memoria.

Hay una forma alternativa de inicializar vectores. En este fragmento se definen e inicializan dos vectores, uno con todos sus elementos a 0 y otro con una secuencia ascendente de números:

```
#define TALLA 5
...
int a[TALLA] = {0, 0, 0, 0, 0};
```

```
int b[TALLA] = {1, 2, 3, 4, 5};
```

Ten en cuenta que, al declarar e inicializar simultáneamente un vector, debes indicar explícitamente los valores del vector y, por tanto, esta aproximación sólo es factible para la inicialización de unos pocos valores.

> ***Omisión de talla en declaraciones con inicialización y otro modo de inicializar***
>
> También puedes declarar e inicializar vectores así:
>
> ```
> int a[] = {0, 0, 0, 0, 0};
> int b[] = {1, 2, 3, 4, 5};
> ```
>
> El compilador deduce que la talla del vector es 5, es decir, el número de valores que aparecen a la derecha del igual. Te recomendamos que, ahora que estás aprendiendo, no uses esta forma de declarar vectores: siempre que puedas, opta por una que haga explícito el tamaño del vector.
>
> En C99 es posible inicializar sólo algunos valores del vector. La sintaxis es un poco enrevesada. Aquí tienes un ejemplo en el que sólo inicializamos el primer y último elementos de un vector de talla 10:
>
> ```
> int a[] = {[0] = 0, [9] = 0};
> ```

2.1.3. Un programa de ejemplo: la criba de Eratóstenes

Vamos a ilustrar lo aprendido desarrollando un sencillo programa que calcule y muestre los números primos menores que N, para un valor de N fijo y determinado en el propio programa. Usaremos un método denominado la criba de Eratóstenes, uno de los algoritmos más antiguos y que debemos a un astrónomo, geógrafo, matemático y filósofo de la antigua Grecia. El método utiliza un vector de N valores booleanos (unos o ceros). Si la celda de índice i contiene el valor 1, consideramos que i es primo, y si no, que no lo es. Inicialmente, todas las celdas excepto la de índice 0 valen 1. Entonces «tachamos» (ponemos un 0 en) las celdas cuyo índice es múltiplo de 2. Acto seguido se busca la siguiente casilla que contiene un 1 y se procede a tachar todas las casillas cuyo índice es múltiplo del índice de esta casilla. Y así sucesivamente. Cuando se ha recorrido completamente el vector, las casillas cuyo índice es primo contienen un 1.

Vamos con una primera versión del programa:

eratostenes.c **eratostenes.c**

```
#include <stdio.h>

#define N 100

int main(void)
{
  int criba[N], i, j;

  /* Inicialización */
  criba[0] = 0;
  for (i=1; i<N; i++)
    criba[i] = 1;

  /* Criba de Eratóstenes */
  for (i=2; i<N; i++)
```

```
16       if (criba[i])
17          for (j=2; i*j<N ; j++)
18             criba[i*j] = 0;
19
20    /* Mostrar los resultados */
21    for (i=0; i<N ; i++)
22       if (criba[i])
23          printf ("%d\n", i);
24
25    return 0;
26 }
```

Observa que hemos tenido que decidir qué valor toma N, pues el vector *criba* debe tener un tamaño conocido en el momento en el que se compila el programa. Si deseamos conocer los, digamos, primos menores que 200, tenemos que modificar la línea 3.

Mejoremos el programa. ¿Es necesario utilizar 4 bytes para almacenar un 0 o un 1? Estamos malgastando memoria. Esta otra versión reduce a una cuarta parte el tamaño del vector *criba*:

eratostenes_1.c — **eratostenes.c**

```
1  #include <stdio.h>
2
3  #define N 100
4
5  int main(void)
6  {
7     char criba[N] ;
8     int i, j;
9
10    /* Inicialización */
11    criba[0] = 0;
12    for (i=1; i<N; i++)
13       criba[i] = 1;
14
15    /* Criba de Eratóstenes */
16    for (i=2; i<N; i++)
17       if (criba[i])
18          for (j=2; i*j<N; j++)
19             criba[i*j] = 0;
20
21    /* Mostrar los resultados */
22    for (i=0; i<N; i++)
23       if (criba[i])
24          printf ("%d\n", i);
25
26    return 0;
27 }
```

Mejor así.

EJERCICIOS

▶ **70** Puedes ahorrar tiempo de ejecución haciendo que *i* tome valores entre 2 y la raíz cuadrada de N. Modifica el programa y comprueba que obtienes el mismo resultado.

2.1.4. Otro programa de ejemplo: estadísticas

Queremos efectuar estadísticas con una serie de valores (las edades de 15 personas), así que vamos a diseñar un programa que nos ayude. En una primera versión, solicitaremos

Optimiza, pero no te pases

C permite optimizar mucho los programas y hacer que estos consuman la menor memoria posible o que se ejecuten a mucha velocidad gracias a una adecuada selección de operaciones. En el programa de la criba de Eratóstenes, por ejemplo, aún podemos reducir más el consumo de memoria: para representar un 1 o un 0 basta un solo bit. Como en un **char** caben 8 bits, podemos proponer esta otra solución:

eratostenes_bit.c

```c
#include <stdio.h>
#include <math.h>

#define N 100

int main(void)
{
  char criba[N/8+1]; // Ocupa unas 8 veces menos que la versión anterior.
  int i, j;

  /* Inicialización */
  criba[0] = 254;      // Pone todos los bits a 1 excepto el primero.
  for (i=1; i<=N/8; i++)
    criba[i] = 255;    // Pone todos los bits a 1.

  /* Criba de Eratóstenes */
  for (i=2; i<N; i++)
    if (criba[i/8] & (1 << (i%8))) // Pregunta si el bit en posición i vale 1.
      for (j=2; i*j<N; j++)
        criba[i*j/8] &= ~(1 << ((i*j) % 8)); // Pone a 0 el bit en posición i*j.

  /* Mostrar los resultados */
  for (i=0; i<N; i++)
    if (criba[i/8] & 1 << (i%8))
      printf("%d\n", i);

  return 0;
}
```

¡Buf! La legibilidad deja mucho que desear. Y no sólo eso: consultar si un determinado bit vale 1 y fijar un determinado bit a 0 resultan ser operaciones más costosas que consultar si el valor de un **char** es 1 o, respectivamente, fijar el valor de un **char** a 0, pues debes hacerlo mediante operaciones de división entera, resto de división entera, desplazamiento, negación de bits y el operador &.

¿Vale la pena reducir la memoria a una octava parte si, a cambio, el programa pierde legibilidad y, además, resulta más lento? No hay una respuesta definitiva a esta pregunta. La única respuesta es: depende. En según qué aplicaciones, puede resultar necesario, en otras no. Lo que no debes hacer, al menos de momento, es obsesionarte con la optimización y complicar innecesariamente tus programas.

las edades de todas las personas y, a continuación, calcularemos y mostraremos por pantalla la edad media, la desviación típica, la moda y la mediana. Las fórmulas para el cálculo de la media y la desviación típica de n elementos son:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n},$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}},$$

donde x_i es la edad del individuo número i.[2] La moda es la edad que más veces aparece (si dos o más edades aparecen muchas veces con la máxima frecuencia, asumiremos que una cualquiera de ellas es la moda). La mediana es la edad tal que el 50% de las edades son inferiores o iguales a ella y el restante 50% son mayores o iguales.

Empezamos por la declaración del vector que albergará las 15 edades y por leer los datos:

edades.c — edades.c

```
#include <stdio.h>

#define PERSONAS 15

int main(void)
{
  int edad[PERSONAS], i;

  /* Lectura de edades */
  for (i=0; i<PERSONAS; i++) {
    printf("Por␣favor,␣introduce␣edad␣de␣la␣persona␣número␣%d:␣", i+1);
    scanf("%d", &edad[i]);
  }

  return 0;
}
```

Vale la pena que te detengas a observar cómo indicamos a *scanf* que lea la celda de índice *i* en el vector *edad*: usamos el operador `&` delante de la expresión *edad*`[`*i*`]`. Es lo que cabía esperar: *edad*`[`*i*`]` es un escalar de tipo *int*, y ya sabes que *scanf* espera su dirección de memoria.

Pasamos ahora a calcular la edad media y la desviación típica (no te ha de suponer dificultad alguna con la experiencia adquirida al aprender Python):

edades_1.c — edades.c

```
#include <stdio.h>
#include <math.h>

#define PERSONAS 15

int main(void)
{
  int edad[PERSONAS], i, suma_edad;
  float suma_desviacion, media, desviacion;

  /* Lectura de edades */
  for (i=0; i<PERSONAS; i++) {
    printf("Por␣favor,␣introduce␣edad␣de␣la␣persona␣número␣%d:␣", i+1);
    scanf("%d", &edad[i]);
  }

  /* Cálculo de la media */
  suma_edad = 0;
  for (i=0; i<PERSONAS; i++)
    suma_edad += edad[i];
  media = suma_edad / (float) PERSONAS;

  /* Cálculo de la desviacion típica */
  suma_desviacion = 0.0;
```

[2]Hay una definición alternativa de la desviación típica en la que el denominador de la fracción es $n-1$.

```
  for (i=0; i<PERSONAS; i++)
    suma_desviacion += (edad[i] - media) * (edad[i] - media);
  desviacion = sqrt( suma_desviacion / PERSONAS );

  /* Impresión de resultados */
  printf("Edad␣media␣␣:␣%f\n", media);
  printf("Desv.␣típica:␣%f\n", desviacion);

  return 0;
}
```

El cálculo de la moda (la edad más frecuente) resulta más problemática. ¿Cómo abordar el cálculo? Vamos a presentar dos versiones diferentes. Empezamos por una que consume demasiada memoria. Dado que trabajamos con edades, podemos asumir que ninguna edad iguala o supera los 150 años. Podemos crear un vector con 150 contadores, uno para cada posible edad:

edades_2.c **edades.c**

```
#include <stdio.h>
#include <math.h>

#define PERSONAS 15
#define MAX_EDAD 150

int main(void)
{
  int edad[PERSONAS], i, suma_edad;
  float suma_desviacion, media, desviacion;
  int contador[MAX_EDAD], frecuencia, moda;

  /* Lectura de edades */
  for (i=0; i<PERSONAS; i++) {
    printf("Por␣favor,␣introduce␣edad␣de␣la␣persona␣número␣%d:␣", i+1);
    scanf("%d", &edad[i]);
  }

  /* Cálculo de la media */
  suma_edad = 0;
  for (i=0; i<PERSONAS; i++)
    suma_edad += edad[i];
  media = suma_edad / (float) PERSONAS;

  /* Cálculo de la desviacion típica */
  suma_desviacion = 0.0;
  for (i=0; i<PERSONAS; i++)
    suma_desviacion += (edad[i] - media) * (edad[i] - media);
  desviacion = sqrt( suma_desviacion / PERSONAS );

  /* Cálculo de la moda */
  for (i=0; i<MAX_EDAD; i++)  // Inicialización de los contadores.
    contador[i] = 0;
  for (i=0; i<PERSONAS; i++)
    contador[edad[i]]++;      // Incrementamos el contador asociado a edad[i].
  moda = -1;
  frecuencia = 0;
  for (i=0; i<MAX_EDAD; i++)  // Buscamos la moda (edad con mayor valor en contador).
    if (contador[i] > frecuencia) {
      frecuencia = contador[i];
      moda = i;
    }
```

```

   /* Impresión de resultados */
   printf ("Edad␣media␣␣:␣%f\n", media);
   printf ("Desv.␣típica:␣%f\n", desviacion);
   printf ("Moda␣␣␣␣␣␣␣␣:␣%d\n", moda);

   return 0;
}
```

Esta solución consume un vector de 150 elementos enteros cuando no es estrictamente necesario. Otra posibilidad pasa por ordenar el vector de edades y contar la longitud de cada secuencia de edades iguales. La edad cuya secuencia sea más larga es la moda:

edades_3.c   **edades.c**

```
#include <stdio.h>
#include <math.h>

#define PERSONAS 15

int main(void)
{
   int edad[PERSONAS], i, j, aux, suma_edad;
   float suma_desviacion, media, desviacion;
   int moda, frecuencia, frecuencia_moda;

   /* Lectura de edades */
   for (i=0; i<PERSONAS; i++) {
      printf ("Por␣favor,␣introduce␣edad␣de␣la␣persona␣número␣%d:␣", i+1);
      scanf ("%d", &edad[i]);
   }

   /* Cálculo de la media */
   suma_edad = 0;
   for (i=0; i<PERSONAS; i++)
      suma_edad += edad[i];
   media = suma_edad / (float) PERSONAS;

   /* Cálculo de la desviacion típica */
   suma_desviacion = 0.0;
   for (i=0; i<PERSONAS; i++)
      suma_desviacion += (edad[i] - media) * (edad[i] - media);
   desviacion = sqrt( suma_desviacion / PERSONAS );

   /* Cálculo de la moda */
   for (i=0; i<PERSONAS-1; i++)  // Ordenación mediante burbuja.
      for (j=0; j<PERSONAS-i; j++)
         if (edad[j] > edad[j+1]) {
            aux = edad[j];
            edad[j] = edad[j+1];
            edad[j+1] = aux;
         }

   frecuencia = 0;
   frecuencia_moda = 0;
   moda = -1;
   for (i=0; i<PERSONAS-1; i++)  // Búsqueda de la serie de valores idénticos más larga.
      if (edad[i] == edad[i+1]) {
         frecuencia++;
         if (frecuencia > frecuencia_moda) {
            frecuencia_moda = frecuencia;
```

```
            moda = edad[i];
         }
      }
      else
         frecuencia = 0;

   /* Impresión de resultados */
   printf("Edad␣media␣␣:␣%f\n", media);
   printf("Desv.␣típica:␣%f\n", desviacion);
   printf("Moda␣␣␣␣␣␣␣␣:␣%d\n", moda);

   return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 71 ¿Contiene en cada instante la variable *frecuencia* el verdadero valor de la frecuencia de aparición de un valor? Si no es así, ¿qué contiene? ¿Afecta eso al cálculo efectuado? ¿Por qué?

▶ 72 Esta nueva versión del programa presenta la ventaja adicional de no fijar un límite máximo a la edad de las personas. El programa resulta, así, de aplicación más general. ¿Son todo ventajas? ¿Ves algún aspecto negativo? Reflexiona sobre la velocidad de ejecución del programa comparada con la del programa que consume más memoria.

. . . .

Sólo nos resta calcular la mediana. Mmmm. No hay que hacer nuevos cálculos para conocer la mediana: gracias a que hemos ordenado el vector, la mediana es el valor que ocupa la posición central del vector, es decir, la edad de índice PERSONAS/2.

edades_4.c **edades.c**

```
#include <stdio.h>
#include <math.h>

#define PERSONAS 15

int main(void)
{
   int edad[PERSONAS], i, j, aux, suma_edad;
   float suma_desviacion, media, desviacion;
   int moda, frecuencia, frecuencia_moda, mediana;

   /* Lectura de edades */
   for (i=0; i<PERSONAS; i++) {
      printf("Por␣favor,␣introduce␣edad␣de␣la␣persona␣número␣%d:␣", i+1);
      scanf("%d", &edad[i]);
   }

   /* Cálculo de la media */
   suma_edad = 0;
   for (i=0; i<PERSONAS; i++)
      suma_edad += edad[i];
   media = suma_edad / (float) PERSONAS;

   /* Cálculo de la desviacion típica */
   suma_desviacion = 0.0;
   for (i=0; i<PERSONAS; i++)
      suma_desviacion += (edad[i] - media) * (edad[i] - media);
   desviacion = sqrt( suma_desviacion / PERSONAS );

```

```
30    /* Cálculo de la moda */
31    for (i=0; i<PERSONAS-1; i++)   // Ordenación mediante burbuja.
32      for (j=0; j<PERSONAS-i; j++)
33        if (edad[j] > edad[j+1]) {
34          aux = edad[j];
35          edad[j] = edad[j+1];
36          edad[j+1] = aux;
37        }
38
39    frecuencia = 0;
40    frecuencia_moda = 0;
41    moda = -1;
42    for (i=0; i<PERSONAS-1; i++)
43      if (edad[i] == edad[i+1])
44        if (++frecuencia > frecuencia_moda) { // Ver ejercicio 73.
45          frecuencia_moda = frecuencia;
46          moda = edad[i];
47        }
48      else
49        frecuencia = 0;
50
51    /* Cálculo de la mediana */
52    mediana = edad[PERSONAS/2]
53
54    /* Impresión de resultados */
55    printf("Edad␣media␣␣:␣%f\n", media);
56    printf("Desv.␣típica:␣%f\n", desviacion);
57    printf("Moda␣␣␣␣␣␣␣␣:␣%d\n", moda);
58    printf("Mediana␣␣␣␣␣:␣%d\n", mediana);
59
60    return 0;
61  }
```

. . . . EJERCICIOS . . . .

▶ **73** Fíjate en la línea 44 del programa y compárala con las líneas 44 y 45 de su versión anterior. ¿Es correcto ese cambio? ¿Lo sería este otro?:

```
44        if (frecuencia++ > frecuencia_moda) {
```

. . . .

Bueno, vamos a modificar ahora el programa para que el usuario introduzca cuantas edades desee hasta un máximo de 20. Cuando se introduzca un valor negativo para la edad, entenderemos que ha finalizado la introducción de datos.

edades.c

```
#include <stdio.h>
#include <math.h>

#define PERSONAS 20

int main(void)
{
  int edad[PERSONAS], i, j, aux, suma_edad;
  float suma_desviacion, media, desviacion;
  int moda, frecuencia, frecuencia_moda, mediana;

  /* Lectura de edades */
  for (i=0; i<PERSONAS; i++) {
    printf("Introduce␣edad␣de␣%d␣(si␣es␣negativa,␣acaba):␣", i+1);
    scanf("%d", &edad[i]);
```

```
      if (edad[i] < 0)
        break;
    }

    ...

    return 0;
}
```

Mmmm. Hay un problema: si no damos 20 edades, el vector presentará toda una serie de valores sin inicializar y, por tanto, con valores arbitrarios. Sería un grave error tomar esos valores por edades introducidas por el usuario. Una buena idea consiste en utilizar una variable entera que nos diga en todo momento cuántos valores introdujo realmente el usuario en el vector *edad*:

edades_5.c **edades.c**

```
#include <stdio.h>
#include <math.h>

#define MAX_PERSONAS 20

int main(void)
{
    int edad[MAX_PERSONAS], personas, i, j, aux, suma_edad;
    float suma_desviacion, media, desviacion;
    int moda, frecuencia, frecuencia_moda, mediana;

    /* Lectura de edades */
    personas = 0;
    for (i=0; i<MAX_PERSONAS; i++) {
      printf("Introduce␣edad␣de␣%d␣(si␣es␣negativa,␣acabar):␣", i+1);
      scanf("%d", &edad[i]);
      if (edad[i] < 0)
        break;
      personas++;
    }

    ...

    return 0;
}
```

La constante que hasta ahora se llamaba PERSONAS ha pasado a llamarse MAX_PERSONAS. Se pretende reflejar que su valor es la *máxima* cantidad de edades de personas que podemos manejar, pues el número de edades que manejamos realmente pasa a estar en la variable entera *personas*.

Una forma alternativa de hacer lo mismo nos permite prescindir del índice *i*:

edades_6.c **edades.c**

```
#include <stdio.h>
#include <math.h>

#define MAX_PERSONAS 20

int main(void)
{
    int edad[MAX_PERSONAS], personas, j, aux, suma_edad;
    float suma_desviacion, media, desviacion;
    int moda, frecuencia, frecuencia_moda, mediana;
```

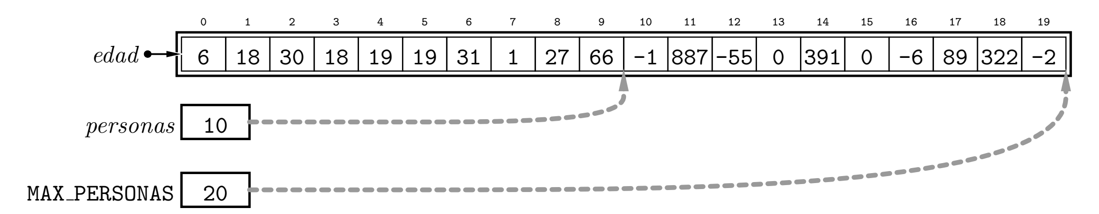
```c

  /* Lectura de edades */
  personas = 0;
  do {
    printf("Introduce edad de %d (si es negativa, acabar): ", personas+1);
    scanf("%d", &edad[personas]);
    personas++;
  } while (personas < MAX_PERSONAS && edad[personas-1] >= 0);
  personas--;

  ...

  return 0;
}
```

Imagina que se han introducido edades de 10 personas. La variable *personas* apunta (conceptualmente) al final de la serie de valores que hemos de considerar para efectuar los cálculos pertinentes:



Ya podemos calcular la edad media, pero con un cuidado especial por las posibles divisiones por cero que provocaría que el usuario escribiera una edad negativa como edad de la primera persona (en cuyo caso *personas* valdría 0):

edades_7.c

edades.c

```c
#include <stdio.h>
#include <math.h>

#define MAX_PERSONAS 20

int main(void)
{
  int edad[MAX_PERSONAS], personas, i, j, aux, suma_edad;
  float suma_desviacion, media, desviacion;
  int moda, frecuencia, frecuencia_moda, mediana;

  /* Lectura de edades */
  personas = 0;
  do {
    printf("Introduce edad de %d (si es negativa, acabar): ", personas+1);
    scanf("%d", &edad[personas]);
    personas++;
  } while (personas < MAX_PERSONAS && edad[personas-1] >= 0);
  personas--;

  if (personas > 0) {
    /* Cálculo de la media */
    suma_edad = 0;
    for (i=0; i<personas; i++)
      suma_edad += edad[i];
    media = suma_edad / (float) personas;

    /* Cálculo de la desviacion típica */
```

```
    suma_desviacion = 0.0;
    for (i=0; i<personas; i++)
      suma_desviacion += (edad[i] - media) * (edad[i] - media);
    desviacion = sqrt( suma_desviacion / personas );

    /* Cálculo de la moda */
    for (i=0; i<personas-1; i++)   // Ordenación mediante burbuja.
      for (j=0; j<personas-i; j++)
        if (edad[j] > edad[j+1]) {
          aux = edad[j];
          edad[j] = edad[j+1];
          edad[j+1] = aux;
        }

    frecuencia = 0;
    frecuencia_moda = 0;
    moda = -1;
    for (i=0; i<personas-1; i++)
      if (edad[i] == edad[i+1])
        if (++frecuencia > frecuencia_moda) {
          frecuencia_moda = frecuencia;
          moda = edad[i];
        }
      else
        frecuencia = 0;

    /* Cálculo de la mediana */
    mediana = edad[personas/2];

    /* Impresión de resultados */
    printf("Edad␣media␣␣:␣%f\n", media);
    printf("Desv.␣típica:␣%f\n", desviacion);
    printf("Moda␣␣␣␣␣␣␣␣:␣%d\n", moda);
    printf("Mediana␣␣␣␣␣:␣%d\n", mediana);
  }
  else
   printf("No␣se␣introdujo␣dato␣alguno.\n");

  return 0;
}
```

EJERCICIOS

▶ 74 Cuando el número de edades es par no hay elemento central en el vector ordenado, así que estamos escogiendo la mediana como uno cualquiera de los elementos «centrales». Utiliza una definición alternativa de edad mediana que considera que su valor es la media de las dos edades que ocupan las posiciones más próximas al centro.

▶ 75 Modifica el ejercicio anterior para que, caso de haber dos o más valores con la máxima frecuencia de aparición, se muestren todos por pantalla al solicitar la moda.

▶ 76 Modifica el programa anterior para que permita efectuar cálculos con hasta 100 personas.

▶ 77 Modifica el programa del ejercicio anterior para que muestre, además, cuántas edades hay entre 0 y 9 años, entre 10 y 19, entre 20 y 29, etc. Considera que ninguna edad es igual o superior a 150.

Ejemplo: si el usuario introduce las siguientes edades correspondientes a 12 personas:

```
10 23 15 18 20 18 57 12 29 31 78 28
```

el programa mostrará (además de la media, desviación típica, moda y mediana), la siguiente tabla:

```
  0 -   9: 0
 10 -  19: 5
 20 -  29: 4
 30 -  39: 1
 40 -  49: 0
 50 -  59: 1
 60 -  69: 0
 70 -  79: 1
 80 -  89: 0
 90 -  99: 0
100 - 109: 0
110 - 119: 0
120 - 129: 0
130 - 139: 0
140 - 149: 0
```

▶ **78** Modifica el programa para que muestre un histograma de edades. La tabla anterior se mostrará ahora como este histograma:

```
  0 -   9:
 10 -  19: *****
 20 -  29: ****
 30 -  39: *
 40 -  49:
 50 -  59: *
 60 -  69:
 70 -  79: *
 80 -  89:
 90 -  99:
100 - 109:
110 - 119:
120 - 129:
130 - 139:
140 - 149:
```

Como puedes ver, cada asterisco representa la edad de una persona.

▶ **79** Modifica el programa anterior para que el primer y último rangos de edades mostrados en el histograma correspondan a tramos de edades en los que hay al menos una persona. El histograma mostrado antes aparecerá ahora así:

```
 10 -  19: *****
 20 -  29: ****
 30 -  39: *
 40 -  49:
 50 -  59: *
 60 -  69:
 70 -  79: *
```

▶ **80** Modifica el programa del ejercicio anterior para que muestre el mismo histograma de esta otra forma:

```
| ######### |           |           |           |           |           |           |
| ######### | ######### |           |           |           |           |           |
| ######### | ######### |           |           |           |           |           |
| ######### | ######### |           |           |           |           |           |
| ######### | ######### | ######### |           | ######### |           | ######### |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|  10 -  19 |  20 -  29 |  30 -  39 |  40 -  49 |  50 -  59 |  60 -  69 |  70 -  79 |
```

. . . .

2.1.5. Otro programa de ejemplo: una calculadora para polinomios

Deseamos implementar una calculadora para polinomios de grado menor o igual que 10. Un polinomio $p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + \cdots + p_{10}x^{10}$ puede representarse con un vector en el que se almacenan sus 11 coeficientes (p_0, p_1, ..., p_{10}). Vamos a construir un programa C que permita leer por teclado dos polinomios $p(x)$ y $q(x)$ y, una vez leídos, calcule los polinomios $s(x) = p(x) + q(x)$ y $m(x) = p(x) \cdot q(x)$.

Empezaremos definiendo dos vectores p y q que han de poder contener 11 valores en coma flotante:

polinomios.c

```c
#include <stdio.h>
#define TALLA_POLINOMIO 11

int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO];
  ...
```

Como leer por teclado 11 valores para p y 11 más para q es innecesario cuando trabajamos con polinomios de grado menor que 10, nuestro programa leerá los datos pidiendo en primer lugar el grado de cada uno de los polinomios y solicitando únicamente el valor de los coeficientes de grado menor o igual que el indicado:

polinomios.c

```c
#include <stdio.h>

#define TALLA_POLINOMIO 11

int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO];
  int grado;
  int i;

  /* Lectura de p */
  do {
    printf("Grado de p (entre 0 y %d): ", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("p_%d: ", i); scanf("%f", &p[i]);
  }

  /* Lectura de q */
  do {
    printf("Grado de q (entre 0 y %d): ", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("q_%d: ", i); scanf("%f", &q[i]);
  }

  return 0;
}
```

El programa presenta un problema: no inicializa los coeficientes que correponden a los términos x^n, para n mayor que el grado del polinomio. Como dichos valores deben

ser nulos, hemos de inicializarlos explícitamente (en aras de la brevedad mostramos únicamente la inicialización de los coeficientes de p):

polinomios.c

```
...
int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO];
  int grado;
  int i;

  /* Lectura de p */
  do {
    printf("Grado de p (entre 0 y %d): ", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("p_%d: ", i); scanf("%f", &p[i]);
  }
  for (i=grado+1; i<TALLA_POLINOMIO; i++)
    p[i] = 0.0;
  ...
  return 0;
}
```

Ahora que hemos leído los polinomios, calculemos la suma. La almacenaremos en un nuevo vector llamado *s*. La suma de dos polinomios de grado menor que TALLA_POLINOMIO es un polinomio de grado también menor que TALLA_POLINOMIO, así que el vector *s* tendrá talla TALLA_POLINOMIO.

polinomios.c

```
...
int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO], s[TALLA_POLINOMIO];
  ...
```

El procedimiento para calcular la suma de polinomios es sencillo. He aquí el cálculo y la presentación del resultado en pantalla:

polinomios.c

polinomios.c

```
#include <stdio.h>

#define TALLA_POLINOMIO 11

int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO], s[TALLA_POLINOMIO];
  int grado;
  int i;

  /* Lectura de p */
  do {
    printf("Grado de p (entre 0 y %d): ", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("p_%d: ", i); scanf("%f", &p[i]);
  }
  for (i=grado+1; i<TALLA_POLINOMIO; i++)
```

```
    p[i] = 0.0;

  /* Lectura de q */
  do {
    printf("Grado␣de␣q␣(entre␣0␣y␣%d):␣", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("q_%d:␣", i); scanf("%f", &q[i]);
  }
  for (i=grado+1; i<TALLA_POLINOMIO; i++)
    q[i] = 0.0;

  /* Cálculo de la suma */
  for (i=0; i<TALLA_POLINOMIO; i++)
    s[i] = p[i] + q[i];

  /* Presentación del resultado */
  printf("Suma:␣%f␣", s[0]);
  for (i=1; i<TALLA_POLINOMIO; i++)
    printf("+␣%f␣x^%d␣", s[i], i);
  printf("\n");

  return 0;
}
```

Aquí tienes un ejemplo de uso del programa con los polinomios $p(x) = 5 + 3x + 5x^2 + x^3$ y $q(x) = 4 - 4x - 5x^2 + x^3$:

```
Grado de p (entre 0 y 10): 3 ↲
p_0: 5 ↲
p_1: 3 ↲
p_2: 5 ↲
p_3: 1 ↲
Grado de q (entre 0 y 10): 3 ↲
q_0: 4 ↲
q_1: -4 ↲
q_2: -5 ↲
q_3: 1 ↲
Suma: 9.000000 + -1.000000 x^1 + 0.000000 x^2 + 2.000000 x^3 + 0.000000 x^4
+ 0.000000 x^5 + 0.000000 x^6 + 0.000000 x^7 + 0.000000 x^8 + 0.000000 x^9
+ 0.000000 x^10
```

EJERCICIOS

▶ 81 Modifica el programa anterior para que no se muestren los coeficientes nulos.

▶ 82 Tras efectuar los cambios propuestos en el ejercicio anterior no aparecerá nada por pantalla cuando todos los valores del polinomio sean nulos. Modifica el programa para que, en tal caso, se muestre por pantalla `0.000000`.

▶ 83 Tras efectuar los cambios propuestos en los ejercicios anteriores, el polinomio empieza con un molesto signo positivo cuando s_0 es nulo. Corrige el programa para que el primer término del polinomio no sea precedido por el carácter `+`.

▶ 84 Cuando un coeficiente es negativo, por ejemplo −1, el programa anterior muestra su correspondiente término en pantalla así: `+ -1.000 x^1`. Modifica el programa anterior para que un término con coeficiente negativo como el del ejemplo se muestre así: `- 1.000000 x^1`.

. . . .

Nos queda lo más difícil: el producto de los dos polinomios. Lo almacenaremos en un vector llamado m. Como el producto de dos polinomios de grado menor o igual que n es un polinomio de grado menor o igual que $2n$, la talla del vector m no es TALLA_POLINOMIO:

```
...
int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO], s[TALLA_POLINOMIO];
  float m[2*TALLA_POLINOMIO-1];
  ...
```

. . . . EJERCICIOS . . . .

▶ **85** ¿Entiendes por qué hemos reservado 2*TALLA_POLINOMIO-1 elementos para m y no 2*TALLA_POLINOMIO?

. . . .

El coeficiente m_i, para valores de i entre 0 y el grado máximo de $m(x)$, es decir, entre los enteros 0 y 2*TALLA_POLINOMIO-2, se calcula así:

$$m_i = \sum_{j=0}^{i} p_j \cdot q_{i-j}.$$

Deberemos tener cuidado de no acceder erróneamente a elementos de p o q fuera del rango de índices válidos.

Implementemos ese cálculo:

polinomios_1.c **polinomios.c**

```
#include <stdio.h>

#define TALLA_POLINOMIO 11

int main(void)
{
  float p[TALLA_POLINOMIO], q[TALLA_POLINOMIO], s[TALLA_POLINOMIO];
  float m[2*TALLA_POLINOMIO-1];
  int grado;
  int i, j;

  /* Lectura de p */
  do {
    printf("Grado␣de␣p␣(entre␣0␣y␣%d):␣", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("p_%d:␣", i); scanf("%f", &p[i]);
  }
  for (i=grado+1; i<TALLA_POLINOMIO; i++)
    p[i] = 0.0;

  /* Lectura de q */
  do {
    printf("Grado␣de␣q␣(entre␣0␣y␣%d):␣", TALLA_POLINOMIO-1);
    scanf("%d", &grado);
  } while (grado < 0 || grado >= TALLA_POLINOMIO);
  for (i = 0; i<=grado; i++) {
    printf("q_%d:␣", i); scanf("%f", &q[i]);
```

```
  }
  for (i=grado+1; i<TALLA_POLINOMIO; i++)
    q[i] = 0.0;

  /* Cálculo de la suma */
  for (i=0; i<TALLA_POLINOMIO; i++)
    s[i] = p[i] + q[i];

  /* Presentación del resultado */
  printf("Suma:␣%f␣", s[0]);
  for (i=1; i<TALLA_POLINOMIO; i++)
    printf("+␣%f␣x^%d␣", s[i], i);
  printf("\n");

  /* Cálculo del producto */
  for (i=0; i<2*TALLA_POLINOMIO-1; i++) {
    m[i] = 0.0;
    for (j=0; j<=i; j++)
      if (j < TALLA_POLINOMIO && i-j < TALLA_POLINOMIO)
        m[i] += p[j] * q[i-j];
  }

  /* Presentación del resultado */
  printf("Producto:␣%f␣", m[0]);
  for (i=1; i<2*TALLA_POLINOMIO-1; i++)
    printf("+␣%f␣x^%d␣", m[i], i);
  printf("\n");

  return 0;
}
```

Observa que nos hubiera venido bien definir sendas funciones para la lectura y escritura de los polinomios, pero al no saber definir funciones todavía, hemos tenido que copiar dos veces el fragmento de programa correspondiente.

. . . . EJERCICIOS . . . .

▶ 86 El programa que hemos diseñado es ineficiente. Si, por ejemplo, trabajamos con polinomios de grado 5, sigue operando con los coeficientes correspondientes a x^6, x^7,..., x^{10}, que son nulos. Modifica el programa para que, con la ayuda de variables enteras, recuerde el grado de los polinomios $p(x)$ y $q(x)$ en sendas variables *talla_p* y *talla_q* y use esta información en los cálculos de modo que se opere únicamente con los coeficientes de los términos de grado menor o igual que el grado del polinomio.

. . . .

Ahora que hemos presentado tres programas ilustrativos del uso de vectores en C, fíjate en que:

- El tamaño de los vectores siempre se determina en tiempo de compilación.
- En un vector podemos almacenar una cantidad de elementos menor o igual que la declarada en su capacidad, nunca mayor.
- Si almacenamos menos elementos de los que caben (como en el programa que efectúa estadísticas de una serie de edades), necesitas alguna variable auxiliar que te permita saber en todo momento cuántas de las celdas contienen información. Si añades un elemento, has de incrementar tú mismo el valor de esa variable.

Ya sabes lo suficiente sobre vectores para poder hacer frente a estos ejercicios:

▶ **87** Diseña un programa que pida el valor de 10 números enteros *distintos* y los almacene en un vector. Si se da el caso, el programa advertirá al usuario, tan pronto sea posible, si introduce un número repetido y solicitará nuevamente el número hasta que sea diferente de todos los anteriores. A continuación, el programa mostrará los 10 números por pantalla.

▶ **88** En una estación meteorológica registramos la temperatura (en grados centígrados) cada hora durante una semana. Almacenamos el resultado en un vector de 168 componentes (que es el resultado del producto 7×24). Diseña un programa que lea los datos por teclado y muestre:

- La máxima y mínima temperaturas de la semana.
- La máxima y mínima temperaturas de cada día.
- La temperatura media de la semana.
- La temperatura media de cada día.
- El número de días en los que la temperatura media fue superior a 30 grados.
- El día y hora en que se registró la mayor temperatura.

▶ **89** La cabecera `stdlib.h` incluye la declaración de funciones para generar números aleatorios. La función *rand*, que no tiene parámetros, devuelve un entero positivo aleatorio. Si deseas generar números aleatorios entre 0 y un valor dado `N`, puedes evaluar *rand*`() % (N+1)`. Cuando ejecutas un programa que usa *rand*, la semilla del generador de números aleatorios es siempre la misma, así que acabas obteniendo la misma secuencia de números aleatorios. Puedes cambiar la semilla del generador de números aleatorios pasándole a la función *srand* un número entero sin signo.

Usa el generador de números aleatorios para inicializar un vector de 10 elementos con números enteros entre 0 y 4. Muestra por pantalla el resultado. Detecta y muestra, a continuación, el tamaño de la sucesión más larga de números consecutivos iguales.

(Ejemplo: si los números generados son `0 4 3 3 2 1 3 2 2 2`, el tramo más largo formado por números iguales es de talla 3 (los tres doses al final de la secuencia), así que por pantalla aparecerá el valor 3.)

▶ **90** Modifica el ejercicio anterior para que trabaje con un vector de 100 elementos.

▶ **91** Genera un vector con 20 números aleatorios entre 0 y 100 y muestra por pantalla el vector resultante y la secuencia de números crecientes consecutivos más larga.

(Ejemplo: la secuencia `1 33 73 85 87 93 99` es la secuencia creciente más larga en la serie de números `87 45 34 12 1 33 73 85 87 93 99 0 100 65 32 17 29 16 12 0`.)

▶ **92** Escribe un programa C que ejecute 1000 veces el cálculo de la longitud de la secuencia más larga sobre diferentes secuencias aleatorias (ver ejercicio anterior) y que muestre la longitud media y desviación típica de dichas secuencias.

▶ **93** Genera 100 números aleatorios entre 0 y 1000 y almacénalos en un vector. Determina a continuación qué números aparecen más de una vez.

▶ **94** Genera 100 números aleatorios entre 0 y 10 y almacénalos en un vector. Determina a continuación cuál es el número que aparece más veces.

▶ **95** Diseña un programa C que almacene en un vector los 100 primeros números primos.

▶ **96** Diseña un programa C que lea y almacene en un vector 10 números enteros asegurándose de que sean positivos. A continuación, el programa pedirá que se introduzca una serie de números enteros y nos dirá si cada uno de ellos está o no en el vector. El programa finaliza cuando el usuario introduce un número negativo.

▶ **97** Diseña un programa C que lea y almacene en un vector 10 números enteros asegurándose de que sean positivos. A continuación, el programa pedirá que se introduzca una serie de números enteros y nos dirá si cada uno de ellos está o no en el vector. El programa finaliza cuando el usuario introduce un número negativo.

Debes ordenar por el método de la burbuja el vector de 10 elementos tan pronto se conocen sus valores. Cuando debas averiguar si un número está o no en el vector, utiliza el algoritmo de búsqueda dicotómica.

. . . .

2.1.6. Disposición de los vectores en memoria

Es importante que conozcas bien cómo se disponen los vectores en memoria. Cuando se encuentra esta declaración en un programa:

```
int a[5];
```

el compilador reserva una zona de memoria contigua capaz de albergar 5 valores de tipo **int**. Como una variable de tipo **int** ocupa 4 bytes, el vector *a* ocupará 20 bytes.

Podemos comprobarlo con este programa:

```
#include <stdio.h>

#define TALLA 5

int main(void)
{
  int a[TALLA];

  printf("Ocupación␣de␣un␣elemento␣de␣a␣(en␣bytes):␣%d\n", sizeof(a[0]));
  printf("Ocupación␣de␣a␣(en␣bytes):␣%d\n", sizeof(a));
  return 0;
}
```

El resultado de ejecutarlo es éste:

```
Ocupación de un elemento de a (en bytes): 4
Ocupación de a (en bytes): 20
```

Cada byte de la memoria tiene una dirección. Si, pongamos por caso, el vector *a* empieza en la dirección 1000, *a*[0] se almacena en los bytes 1000–1003, *a*[1] en los bytes 1004–1007, y así sucesivamente. El último elemento, *a*[4], ocupará los bytes 1016–1019:

996:					
1000:					*a*[0]
1004:					*a*[1]
1008:					*a*[2]
1012:					*a*[3]
1016:					*a*[4]
1020:					

Big-endian y little-endian

Lo bueno de los estándares es... que hay muchos donde elegir. No hay forma de ponerse de acuerdo. Muchos ordenadores almacenan los números enteros de más de 8 bits disponiendo los bits más significativos en la dirección de memoria más baja y otros, en la más alta. Los primeros se dice que siguen la codificación «big-endian» y los segundos, «little-endian».

Pongamos un ejemplo. El número 67586 se representa en binario con cuatro bytes:

```
00000000 00000001 00001000 00000010
```

Supongamos que ese valor se almacena en los cuatro bytes que empiezan en la dirección 1000. En un ordenador «big-endian», se dispondrían en memoria así (te indicamos bajo cada byte su dirección de memoria):

1000:	00000000	00000001	00001000	00000010
	1000	1001	1002	1003

En un ordenador «little-endian», por contra, se representaría de esta otra forma:

1000:	00000010	00001000	00000001	00000000
	1000	1001	1002	1003

Los ordenadores PC (que usan microprocesadores Intel y AMD), por ejemplo, son «little-endian» y los Macintosh basados en microprocesadores Motorola son «big-endian». Aunque nosotros trabajamos en clase con ordenadores Intel, te mostraremos los valores binarios como estás acostumbrado a verlos: con el byte más significativo a la izquierda.

La diferente codificación de unas y otras plataformas plantea serios problemas a la hora de intercambiar información en ficheros binarios, es decir, ficheros que contienen volcados de la información en memoria. Nos detendremos nuevamente sobre esta cuestión cuando estudiamos ficheros.

Por cierto, lo de «little-endian» y «big-endian» viene de «Los viajes de Gulliver», la novela de Johnathan Swift. En ella, los liliputienses debaten sobre una importante cuestión política: ¿deben abrirse los huevos pasados por agua por su extremo grande, como defiende el partido Big-Endian, o por su extremo puntiagudo, como mantiene el partido Little-Endian?

¿Recuerdas el operador & que te presentamos en el capítulo anterior? Es un operador unario que permite conocer la dirección de memoria de una variable. Puedes aplicar el operador & a un elemento del vector. Por ejemplo, &*a*[2] es la dirección de memoria en la que empieza *a*[2], es decir, la dirección 1008 en el ejemplo.

Veamos qué dirección ocupa cada elemento de un vector cuando ejecutamos un programa sobre un computador real:

direcciones_vector.c

```c
#include <stdio.h>

#define TALLA 5

int main(void)
{
  int a[TALLA], i;

  for (i = 0; i < TALLA; i++)
    printf("Dirección de a[%d]: %u\n", i, (unsigned int) &a[i]);

  return 0;
}
```

Al ejecutar el programa obtenemos en pantalla lo siguiente (puede que obtengas un resultado diferente si haces la prueba tú mismo, pues el vector puede estar en un lugar cualquiera de la memoria):

```
Dirección de a[0]: 3221222640
Dirección de a[1]: 3221222644
Dirección de a[2]: 3221222648
Dirección de a[3]: 3221222652
Dirección de a[4]: 3221222656
```

¿Ves? Cada dirección de memoria de una celda de *a* se diferencia de la siguiente en 4 unidades.

Recuerda que la función de lectura de datos por teclado *scanf* modifica el valor de una variable cuya dirección de memoria se le suministra. Para depositar en la zona de memoria de la variable el nuevo valor necesita conocer la dirección de memoria. Por esa razón precedíamos los identificadores de las variables con el operador &. Este programa, por ejemplo, lee por teclado el valor de todos los componentes de un vector utilizando el operador & para conocer la dirección de memoria de cada uno de ellos:

lee_vector.c — lee_vector.c

```
#include <stdio.h>

#define TALLA 5

int main(void)
{
  int a[TALLA], i;

  for (i = 0; i < TALLA; i++)
    printf("Introduce␣el␣valor␣de␣a[%d]:", i); scanf("%d", &a[i]);

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 98 ¿Qué problema presenta esta otra versión del mismo programa?

lee_vector_1.c — lee_vector.c

```
#include <stdio.h>

#define TALLA 5

int main(void)
{
  int a[TALLA], i;

  for (i = 0; i < TALLA; i++)
    printf("Introduce␣el␣valor␣de␣a[%d]:", i); scanf("%d", a[i]);

  return 0;
}
```

. . . .

Analiza este programa:

direcciones_vector2.c — direcciones_vector2.c

```
#include <stdio.h>

#define TALLA 5
```

```

int main(void)
{
  int a[TALLA], i;

  for (i = 0; i < TALLA; i++)
    printf("Dirección␣de␣a[%d]:␣%u\n", i, (unsigned int) &a[i]);

  printf("Dirección␣de␣a:␣%u\n", (unsigned int) a);
  return 0;
}
```

He aquí el resultado de ejecutarlo:

```
Dirección de a[0]: 3221222640
Dirección de a[1]: 3221222644
Dirección de a[2]: 3221222648
Dirección de a[3]: 3221222652
Dirección de a[4]: 3221222656
Dirección de a: 3221222640
```

Observa que la dirección de memoria de las líneas primera y última es la misma. En consecuencia, esta línea:

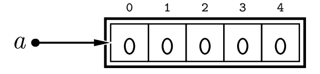
```
printf("Dirección␣de␣a:␣%u\n", (unsigned int) &a[0]);
```

es equivalente a esta otra:

```
printf("Dirección␣de␣a:␣%u\n", (unsigned int) a);
```

Así pues, *a* expresa una dirección de memoria (la de su primer elemento), es decir, *a* es un *puntero* o referencia a memoria y es equivalente a &*a*[0]. La característica de que el identificador de un vector represente, a la vez, al vector y a un puntero que apunta donde empieza el vector recibe el nombre *dualidad vector-puntero*, y es un rasgo propio del lenguaje de programación C.

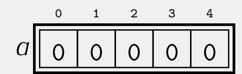
Representaremos esquemáticamente los vectores de modo similar a como representábamos las listas en Python:



Fíjate en que el gráfico pone claramente de manifiesto que *a* es un puntero, pues se le representa con una flecha que apunta a la zona de memoria en la que se almacenan los elementos del vector. Nos interesa diseñar programas con un nivel de abstracción tal que la imagen conceptual que tengamos de los vectores se limite a la del diagrama.

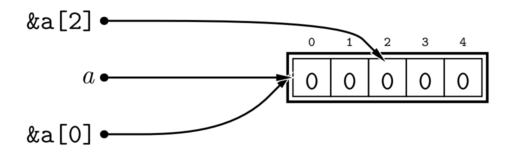
Mentiremos cada vez menos

Lo cierto es que *a* no es exactamente un puntero, aunque funciona como tal. Sería más justo representar la memoria así:



Pero, por el momento, conviene que consideres válida la representación en la que *a* es un puntero. Cuando estudiemos la gestión de memoria dinámica abundaremos en esta cuestión.

Recuerda que el operador & obtiene la dirección de memoria en la que se encuentra un valor. En esta figura te ilustramos &*a*[0] y &*a*[2] como sendos punteros a sus respectivas celdas en el vector.



¿Cómo «encuentra» C la dirección de memoria de un elemento del vector cuando accedemos a través de un índice? Muy sencillo, efectuando un cálculo consistente en sumar al puntero que señala el principio del vector el resultado de multiplicar el índice por el tamaño de un elemento del vector. La expresión *a*[2], por ejemplo, se entiende como «accede al valor de tipo **int** que empieza en la dirección *a* con un desplazamiento de 2×4 bytes». Una sentencia de asignación como *a*[2] = 0 se interpreta como «almacena el valor 0 en el entero **int** que empieza en la dirección de memoria de *a* más 2×4 bytes».

2.1.7. Algunos problemas de C: accesos ilícitos a memoria

Aquí tienes un programa con un resultado que puede sorprenderte:

ilicito.c

```
#include <stdio.h>

#define TALLA 3

int main(void)
{
  int v[TALLA], w[TALLA], i;

  for(i=0; i<TALLA; i++) {
    v[i] = i;
    w[i] = 10 + i;
  }

  printf("+----+-------------+----+\n");
  printf("|␣Objeto␣|␣Dirección␣de␣memoria␣|␣Valor␣|\n");
  printf("+----+-------------+----+\n");
  printf("|␣i␣␣␣␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &i, i);
  printf("+----+-------------+----+\n");
  printf("|␣w[0]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &w[0], w[0]);
  printf("|␣w[1]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &w[1], w[1]);
  printf("|␣w[2]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &w[2], w[2]);
  printf("+----+-------------+----+\n");
  printf("|␣v[0]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[0], v[0]);
  printf("|␣v[1]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[1], v[1]);
  printf("|␣v[2]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[2], v[2]);
  printf("+----+-------------+----+\n");
  printf("|␣v[-2]␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[-2], v[-2]);
  printf("|␣v[-3]␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[-3], v[-3]);
  printf("|␣v[-4]␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[-4], v[-4]);
  printf("|␣w[5]␣␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &w[5], w[5]);
  printf("|␣w[-1]␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &w[-1], w[-1]);
  printf("|␣v[-5]␣␣|␣%20u␣|␣%5d␣|\n", (unsigned int) &v[-5], v[-5]);
  printf("+----+-------------+----+\n");

  return 0;
}
```

Aquí tienes el resultado de su ejecución[3]:

```
+--------+----------------------+-------+
| Objeto | Dirección de memoria | Valor |
+--------+----------------------+-------+
| i      |           3221222636 |     3 |
+--------+----------------------+-------+
| w[0]   |           3221222640 |    10 |
| w[1]   |           3221222644 |    11 |
| w[2]   |           3221222648 |    12 |
+--------+----------------------+-------+
| v[0]   |           3221222656 |     0 |
| v[1]   |           3221222660 |     1 |
| v[2]   |           3221222664 |     2 |
+--------+----------------------+-------+
| v[-2]  |           3221222648 |    12 |
| v[-3]  |           3221222644 |    11 |
| v[-4]  |           3221222640 |    10 |
| w[5]   |           3221222660 |     1 |
| w[-1]  |           3221222636 |     3 |
| v[-5]  |           3221222636 |     3 |
+--------+----------------------+-------+
```

La salida es una tabla con tres columnas: en la primera se indica el objeto que se está estudiando, la segunda corresponde a la dirección de memoria de dicho objeto[4] y la tercera muestra el valor almacenado en dicho objeto. A la vista de las direcciones de memoria de los objetos *i*, *v*[0], *v*[1], *v*[2], *w*[0], *w*[1] y *w*[2], el compilador ha reservado la memoria de estas variables así:

3221222636:	3	i
3221222640:	10	w[0]
3221222644:	11	w[1]
3221222648:	12	w[2]
3221222652:		
3221222656:	0	v[0]
3221222660:	1	v[1]
3221222664:	2	v[2]

Fíjate en que las seis últimas filas de la tabla corresponden a accesos a *v* y *w* con índices fuera de rango. Cuando tratábamos de acceder a un elemento inexistente en una lista Python, el intérprete generaba un error de tipo (error de índice). Ante una situación similar, *C no detecta error alguno*. ¿Qué hace, pues? Aplica la fórmula de indexación, sin más. Estudiemos con calma el primer caso extraño: *v*[-2]. C lo interpreta como: «acceder al valor almacenado en la dirección que resulta de sumar 3221222656 (que es donde empieza el vector *v*) a $(-2) \times 4$ (-2 es el índice del vector y 4 es tamaño de un **int**)». Haz el cálculo: el resultado es 3221222648… ¡la misma dirección de memoria que ocupa el valor de *w*[2]! Esa es la razón de que se muestre el valor 12. En la ejecución del

[3]Nuevamente, una advertencia: puede que obtengas un resultado diferente al ejecutar el programa en tu ordenador. La asignación de direcciones de memoria a cada objeto de un programa es una decisión que adopta el compilador con cierta libertad.

[4]Si ejecutas el programa en tu ordenador, es probable que obtengas valores distintos para las direcciones de memoria. Es normal: en cada ordenador y con cada ejecución se puede reservar una zona de memoria distinta para los datos.

programa, *v*[-2] y *w*[2] son exactamente lo mismo. Encuentra tú mismo una explicación para los restantes accesos ilícitos.

¡Ojo! Que se pueda hacer no significa que sea aconsejable hacerlo. En absoluto. Es más: debes evitar acceder a elementos con índices de vector fuera de rango. Si no conviene hacer algo así, ¿por qué no comprueba C si el índice está en el rango correcto antes de acceder a los elementos y, en caso contrario, nos señala un error? Por eficiencia. Un programa que maneje vectores accederá a sus elementos, muy probablemente, en numerosas ocasiones. Si se ha de comprobar si el índice está en el rango de valores válidos, cada acceso se penalizará con un par de comparaciones y el programa se ejecutará más lentamente. C sacrifica seguridad por velocidad, de ahí que tenga cierta fama (justificadísima) de lenguaje «peligroso».

2.1.8. Asignación y copia de vectores

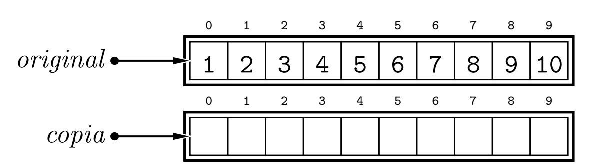
Este programa pretende copiar un vector en otro, pero es incorrecto:

copia_vectores_mal.c — copia_vectores_mal.c

```
1  #define TALLA 10
2
3  int main(void)
4  {
5    int original[TALLA] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
6    int copia[TALLA];
7
8    copia = original;
9
10   return 0;
11 }
```

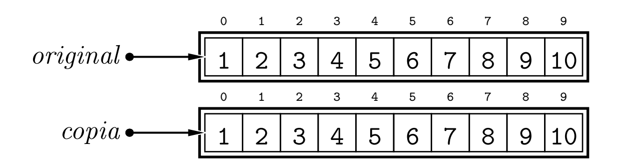
Si compilas el programa, obtendrás un error en la línea 8 que te impedirá obtener un ejecutable: «`incompatible types in assignment`». El mensaje de error nos indica que no es posible efectuar asignaciones entre tipos vectoriales.

Nuestra intención era que antes de ejecutar la línea 8, la memoria presentara este aspecto:

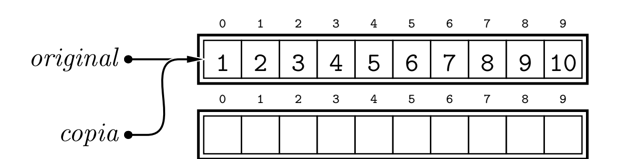


y, una vez ejecutada la línea 8 llegar a una de estas dos situaciones:

1. obtener en *copia* una copia del contenido de *original*:



2. o conseguir que, como en Python, *copia* apunte al mismo lugar que *original*:



Violación de segmento

Los errores de acceso a zonas de memoria no reservada se cuentan entre los peores. En el ejemplo, hemos accedido a la zona de memoria de un vector saliéndonos del rango de indexación válido de otro, lo cual ha producido resultados desconcertantes.

Pero podría habernos ido aún peor: si tratas de escribir en una zona de memoria que no pertenece a ninguna de tus variables, cosa que puedes hacer asignando un valor a un elemento de vector fuera de rango, es posible que se genere una excepción durante la ejecución del programa: intentar escribir en una zona de memoria que no ha sido asignada a nuestro proceso dispara, en Unix, una señal de «violación de segmento» (*segmentation violation*) que provoca la inmediata finalización de la ejecución del programa. Fíjate en este programa:

violacion.c — violacion.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a[10];
6
7    a[10000] = 1;
8
9    return 0;
10 }
```

Cuando lo ejecutamos en un ordenador bajo Unix, obtenemos este mensaje por pantalla:

```
Violación de segmento
```

El programa ha finalizado abruptamente al ejecutar la asignación de la línea 7.

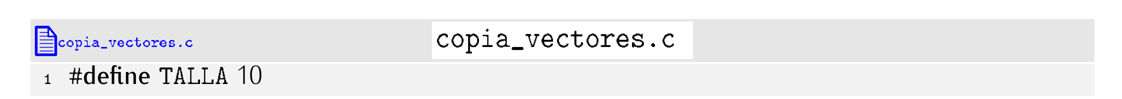
Estos errores en la gestión de memoria se manifiestan de formas muy variadas: pueden producir resultados extraños, finalizar la ejecución incorrectamente o incluso bloquear al computador. ¿Bloquear al computador? Sí, en sistemas operativos poco robustos, como Microsoft Windows, el ordenador puede quedarse bloqueado. (Probablemente has experimentado la sensación usando algunos programas comerciales en el entorno Microsoft Windows.) Ello se debe a que ciertas zonas de memoria deberían estar fuera del alcance de los programas de usuario y el sistema operativo debería prohibir accesos ilícitos. Unix mata al proceso que intenta efectuar accesos ilícitos (de ahí que terminen con mensajes como «Violación de segmento»). Microsoft Windows no tiene la precaución de protegerlas, así que las consecuencias son mucho peores.

Pero casi lo peor es que tu programa puede funcionar mal en unas ocasiones *y bien en otras*. El hecho de que el programa pueda funcionar mal algunas veces y bien el resto es peligrosísimo: como los errores pueden no manifestarse durante el desarrollo del programa, cabe la posibilidad de que no los detectes. Nada peor que dar por bueno un programa que, en realidad, es incorrecto.

Tenlo siempre presente: la gestión de vectores obliga a estar siempre pendiente de no rebasar la zona de memoria reservada.

Pero no ocurre ninguna de las dos cosas: el identificador de un vector estático se considera un puntero *inmutable*. Siempre apunta a la misma dirección de memoria. No puedes asignar un vector a otro porque eso significaría cambiar el valor de su dirección. (Observa, además, que en el segundo caso, la memoria asignada a *copia* quedaría sin puntero que la referenciara.)

Si quieres copiar el contenido de un vector en otro debes hacerlo elemento a elemento:




```
#define TALLA 10
```

```c

int main(void)
{
  int original[TALLA] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
  int copia[TALLA];
  int i;

  for (i=0; i<TALLA; i++)
    copia[i] = original[i];

  return 0;
}
```

2.1.9. Comparación de vectores

En Python podíamos comparar listas. Por ejemplo, [1,2,3] == [1,1+1,3] devolvía *True*. Ya lo habrás adivinado: C no permite comparar vectores. Efectivamente.

Si quieres comparar dos vectores, has de hacerlo elemento a elemento:

compara_vectores.c — compara_vectores.c

```c
#define TALLA 3

int main(void)
{
  int original[TALLA] = { 1, 2, 3 };
  int copia[TALLA] = {1, 1+1, 3};
  int i, son_iguales;

  son_iguales = 1; // Suponemos que todos los elementos son iguales dos a dos.
  i = 0;
  while (i < TALLA && son_iguales) {
    if (copia[i] != original[i]) // Pero basta con que dos elementos no sean iguales...
      son_iguales = 0;           // ... para que los vectores sean distintos.
    i++;
  }

  if (son_iguales)
    printf("Son␣iguales\n");
  else
    printf("No␣son␣iguales\n");

  return 0;
}
```

2.2. Cadenas estáticas

Las cadenas son un tipo de datos básico en Python, pero no en C. Las cadenas de C son vectores de caracteres (elementos de tipo **char**) con una peculiaridad: el texto de la cadena termina siempre en un carácter nulo. El carácter nulo tiene código ASCII 0 y podemos representarlo tanto con el *entero* 0 como con el *carácter* '\0' (recuerda que '\0' es una forma de escribir el valor entero 0). ¡Ojo! No confundas '\0' con '0': el primero vale 0 y el segundo vale 48.

Las cadenas estáticas en C son, a diferencia de las cadenas Python, mutables. Eso significa que puedes modificar el contenido de una cadena durante la ejecución de un programa.

2.2.1. Declaración de cadenas

Las cadenas se declaran como vectores de caracteres, así que debes proporcionar el número máximo de caracteres que es capaz de almacenar: su *capacidad*. Esta cadena, por ejemplo, se declara con capacidad para almacenar 10 caracteres:

```
char a[10];
```

Puedes inicializar la cadena con un valor en el momento de su declaración:

```
char a[10] = "cadena";
```

Hemos declarado *a* como un vector de 10 caracteres y lo hemos inicializado asignándole la cadena `"cadena"`. Fíjate: hemos almacenado en *a* una cadena de menos de 10 caracteres. No hay problema: la *longitud* de la cadena almacenada en *a* es *menor* que la *capacidad* de *a*.

2.2.2. Representación de las cadenas en memoria

A simple vista, `"cadena"` ocupa 6 bytes, pues contamos en ella 6 caracteres, pero no es así. En realidad, `"cadena"` ocupa 7 bytes: los 6 que corresponden a los 6 caracteres que ves más uno correspondiente a *un carácter nulo al final*, que se denomina *terminador de cadena* y es invisible.

Al declarar e inicializar una cadena así:

```
char a[10] = "cadena";
```

la memoria queda de este modo:

```
     0   1   2   3   4   5   6   7   8   9
a → | c | a | d | e | n | a |\0 |   |   |   |
```

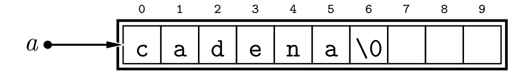
Es decir, es como si hubiésemos inicializado la cadena de este otro modo equivalente:

```
char a[10] = { 'c', 'a', 'd', 'e', 'n', 'a', '\0' };
```

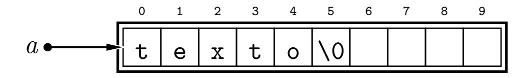
Recuerda, pues, que hay dos valores relacionados con el tamaño de una cadena:

- su *capacidad*, que es la talla del vector de caracteres;
- su *longitud*, que es el número de caracteres que contiene, sin contar el terminador de la cadena. La longitud de la cadena debe ser siempre *estrictamente menor* que la capacidad del vector para no desbordar la memoria reservada.

¿Y por qué toda esta complicación del terminador de cadena? Lo normal al trabajar con una variable de tipo cadena es que su longitud varíe conforme evoluciona la ejecución del programa, pero el tamaño de un vector es fijo. Por ejemplo, si ahora tenemos en *a* el texto `"cadena"` y más tarde decidimos guardar en ella el texto `"texto"`, que tiene un carácter menos, estaremos pasando de esta situación:



a esta otra:



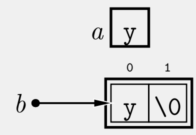
Una cadena de longitud uno no es un carácter

Hemos dicho en el capítulo anterior que una cadena de un sólo carácter, por ejemplo "y", no es lo mismo que un carácter, por ejemplo 'y'. Ahora puedes saber por qué: la diferencia estriba en que "y" ocupa dos bytes, el que corresponde al carácter 'y' y el que corresponde al carácter nulo '\0', mientras que 'y' ocupa un solo byte.

Fíjate en esta declaración de variables:

```
char a = 'y';
char b[2] = "y";
```

He aquí una representación gráfica de las variables y su contenido:

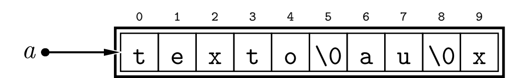


Recuerda:

- Las comillas simples definen un carácter y un carácter ocupa un solo byte.
- Las comillas dobles definen una cadena. Toda cadena incluye un carácter nulo invisible al final.

Fíjate en que la zona de memoria asignada a *a* sigue siendo la misma. El «truco» del terminador ha permitido que la cadena decrezca. Podemos conseguir también que crezca a voluntad... pero siempre que no se rebase la capacidad del vector.

Hemos representado las celdas a la derecha del terminador como cajas vacías, pero no es cierto que lo estén. Lo normal es que contengan valores arbitrarios, aunque eso no importa mucho: el convenio de que la cadena termina en el primer carácter nulo hace que el resto de caracteres no se tenga en cuenta. Es posible que, en el ejemplo anterior, la memoria presente realmente este aspecto:

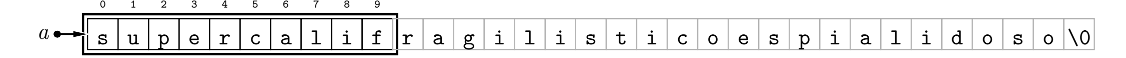


Por comodidad representaremos las celdas a la derecha del terminador con cajas vacías, pues no importa en absoluto lo que contienen.

¿Qué ocurre si intentamos inicializar una zona de memoria reservada para sólo 10 **char**s con una cadena de longitud mayor que 9?

```
char a[10] = "supercalifragilisticoespialidoso"; // ¡Mal!
```

Estaremos cometiendo un gravísimo error de programación que, posiblemente, no detecte el compilador. Los caracteres que no caben en *a* se escriben en la zona de memoria que sigue a la zona ocupada por *a*.



Ya vimos en un apartado anterior las posibles consecuencias de ocupar memoria que no nos ha sido reservada: puede que modifiques el contenido de otras variables o que trates de escribir en una zona que te está vetada, con el consiguiente aborto de la ejecución del programa.

Como resulta que en una variable con capacidad para, por ejemplo, 80 caracteres sólo caben realmente 79 caracteres aparte del nulo, adoptaremos una curiosa práctica al declarar variables de cadena que nos permitirá almacenar los 80 caracteres (además del nulo) sin crear una constante confusión con respecto al número de caracteres que caben en ellas:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1]; /* Reservamos 81 caracteres: 80 caracteres más el terminador */

  return 0;
}
```

2.2.3. Entrada/salida de cadenas

Las cadenas se muestran con *printf* y la adecuada marca de formato sin que se presenten dificultades especiales. Lo que sí resulta problemático es leer cadenas. La función *scanf* presenta una seria limitación: sólo puede leer «palabras», no «frases». Ello nos obligará a presentar una nueva función (*gets*)... que se lleva fatal con *scanf*.

Salida con *printf*

Empecemos por considerar la función *printf*, que muestra cadenas con la marca de formato %s. Aquí tienes un ejemplo de uso:

salida_cadena.c salida_cadena.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1] = "una␣cadena";

  printf("El␣valor␣de␣cadena␣es␣%s.\n", cadena);

  return 0;
}
```

Al ejecutar el programa obtienes en pantalla esto:

```
El valor de cadena es una cadena.
```

Puedes alterar la presentación de la cadena con modificadores:

salida_cadena_con_modificador salida_cadena_con_modificadores.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1] = "una␣cadena";

  printf("El␣valor␣de␣cadena␣es␣(%s).\n", cadena);
```

```
    printf("El␣valor␣de␣cadena␣es␣(%20s).\n", cadena);
    printf("El␣valor␣de␣cadena␣es␣(%-20s).\n", cadena);

    return 0;
}
```

```
El valor de cadena es (una cadena).
El valor de cadena es (          una cadena).
El valor de cadena es (una cadena          ).
```

¿Y si deseamos mostrar una cadena carácter a carácter? Podemos hacerlo llamando a *printf* sobre cada uno de los caracteres, pero recuerda que la marca de formato asociada a un carácter es %c:

salida_caracter_a_caracter.c salida_caracter_a_caracter.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
    char cadena[MAXLON+1] = "una␣cadena";
    int i;

    i = 0;
    while (cadena[i] != '\0') {
        printf("%c\n", cadena[i]);
        i++;
    }

    return 0;
}
```

Este es el resultado de la ejecución:

```
u
n
a

c
a
d
e
n
a
```

Entrada con *scanf*

Poco más hay que contar acerca de *printf*. La función *scanf* es un reto mayor. He aquí un ejemplo que pretende leer e imprimir una cadena en la que podemos guardar hasta 80 caracteres (sin contar el terminador nulo):

lee_una_cadena.c lee_una_cadena.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
```

```
  char cadena[MAXLON+1];

  scanf("%s", cadena);
  printf("La␣cadena␣leída␣es␣%s\n", cadena);

  return 0;
}
```

¡Ojo! ¡No hemos puesto el operador & delante de *cadena*! ¿Es un error? No. *Con las cadenas no hay que poner el carácter & del identificador al usar scanf*. ¿Por qué? Porque *scanf* espera una dirección de memoria y el identificador, por la dualidad vector-puntero, ¡es una dirección de memoria!

Recuerda: *cadena*[0] es un **char**, pero *cadena*, sin más, es la dirección de memoria en la que empieza el vector de caracteres.

Ejecutemos el programa e introduzcamos una palabra:

```
una ↲
La cadena leída es una
```

. . . . EJERCICIOS . . . .

▶ **99** ¿Es válida esta otra forma de leer una cadena? Pruébala en tu ordenador.

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char cadena[MAXLON+1];

  scanf("%s", &cadena[0]);
  printf("La␣cadena␣leída␣es␣%s.\n", cadena);

  return 0;
}
```

. . . .

Cuando *scanf* recibe el valor asociado a *cadena*, recibe una dirección de memoria y, a partir de ella, deja los caracteres leídos de teclado. Debes tener en cuenta que si los caracteres leídos exceden la capacidad de la cadena, se producirá un error de ejecución.

¿Y por qué *printf* no muestra por pantalla una simple dirección de memoria cuando ejecutamos la llamada *printf*("La␣cadena␣leída␣es␣\%s.\n", *cadena*)? Si es cierto lo dicho, *cadena* es una dirección de memoria. La explicación es que la marca %s es interpretada por *printf* como «me pasan una dirección de memoria en la que empieza una cadena, así que he de mostrar su contenido carácter a carácter hasta encontrar un carácter nulo».

Lectura con *gets*

Hay un problema práctico con *scanf*: sólo lee una «palabra», es decir, una secuencia de caracteres no blancos. Hagamos la prueba:

lee_frase_mal.c ϟ lee_frase_mal.c ϟ

```
#include <stdio.h>

#define MAXLON 80

```

```
int main(void)
{
  char cadena[MAXLON+1];

  scanf("%s", cadena);
  printf("La␣cadena␣leída␣es␣%s.\n", cadena);

  return 0;
}
```

Si al ejecutar el programa tecleamos un par de palabras, sólo se muestra la primera:

```
una frase
La cadena leída es una.
```

¿Qué ha ocurrido con los restantes caracteres tecleados? ¡Están a la espera de ser leídos! La siguiente cadena leída, si hubiera un nuevo *scanf*, sería "frase". Si es lo que queríamos, perfecto, pero si no, el desastre puede ser mayúsculo.

¿Cómo leer, pues, una frase completa? No hay forma sencilla de hacerlo con *scanf*. Tendremos que recurrir a una función diferente. La función *gets* lee *todos* los caracteres que hay hasta encontrar un salto de línea. Dichos caracteres, excepto el salto de línea, se almacenan a partir de la dirección de memoria que se indique como argumento y se añade un terminador.

Aquí tienes un ejemplo:

```
#include <stdio.h>

#define MAXLON 11

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];

  printf("Introduce␣una␣cadena:␣"); gets(a);
  printf("Introduce␣otra␣cadena:␣"); gets(b);
  printf("La␣primera␣es␣%s␣y␣la␣segunda␣es␣%s\n", a, b);

  return 0;
}
```

Ejecutemos el programa:

```
Introduce una cadena: uno dos ↲
Introduce otra cadena: tres cuatro ↲
La primera es uno dos y la segunda es tres cuatro
```

Lectura de cadenas y escalares: *gets* y *sscanf*

Y ahora, vamos con un problema al que te enfrentarás en más de una ocasión: la lectura alterna de cadenas y valores escalares. La mezcla de llamadas a *scanf* y a *gets*, produce efectos curiosos que se derivan de la combinación de su diferente comportamiento frente a los blancos. El resultado suele ser una lectura incorrecta de los datos o incluso el bloqueo de la ejecución del programa. Los detalles son bastante escabrosos. Si tienes curiosidad, te los mostramos en el apartado B.3.

Presentaremos en este capítulo una solución directa que deberás aplicar siempre que tu programa alterne la lectura de cadenas con blancos y valores escalares (algo muy frecuente). La solución consiste en:

Overflow exploit

El manejo de cadenas C es complicado... y peligroso. La posibilidad de que se almacenen más caracteres de los que caben en una zona de memoria reservada para una cadena ha dado lugar a una técnica de *cracking* muy común: el *overflow exploit* (que significa «aprovechamiento del desbordamiento»), también conocido por *smash the stack* («machacar la pila»).

Si un programa C lee una cadena con *scanf* o *gets* es vulnerable a este tipo de ataques. La idea básica es la siguiente. Si *c* es una variable local a una función (en el siguiente capítulo veremos cómo), reside en una zona de memoria especial: la pila. Podemos desbordar la zona de memoria reservada para la cadena *c* escribiendo un texto más largo del que cabe en ella. Cuando eso ocurre, estamos ocupando memoria en una zona de la pila que no nos «pertenece». Podemos conseguir así escribir información en una zona de la pila reservada a información como la dirección de retorno de la función. El *exploit* se basa en asignar a la dirección de retorno el valor de una dirección en la que habremos escrito una rutina especial en código máquina. ¿Y cómo conseguimos introducir una rutina en código máquina en un programa ajeno? ¡En la propia cadena que provoca el desbordamiento, codificándola en binario! La rutina de código máquina suele ser sencilla: efectúa una simple llamada al sistema operativo para que ejecute un intérprete de órdenes Unix. El intérprete se ejecutará con los mismos permisos que el programa que hemos reventado. Si el programa atacado se ejecutaba con permisos de `root`, habremos conseguido ejecutar un intérprete de órdenes como `root`. ¡El ordenador es nuestro!

¿Y cómo podemos proteger a nuestros programas de los *overflow exploit*? Pues, para empezar, no utilizando nunca *scanf* o *gets* directamente. Como es posible leer de teclado carácter a carácter (lo veremos en el capítulo dedicado a ficheros), podemos definir nuestra propia función de lectura de cadenas: una función de lectura que controle que nunca se escribe en una zona de memoria más información de la que cabe.

Dado que *gets* es tan vulnerable a los overflow exploit, el compilador de C te dará un aviso cuando la uses. No te sorprendas, pues, cuando veas un mensaje como éste: «`the ‘gets’ function is dangerous and should not be used`».

- Si vas a leer una cadena usar *gets*.
- Y si vas a leer un valor escalar, proceder en dos pasos:
  - leer una línea completa con *gets* (usa una variable auxiliar para ello),
  - y extraer de ella los valores escalares que se deseaba leer con ayuda de la función *sscanf*.

La función *sscanf* es similar a *scanf* (fíjate en la «s» inicial), pero no obtiene información leyéndola del teclado, sino que la extrae de una cadena.

Un ejemplo ayudará a entender el procedimiento:

lecturas.c

```
#include <stdio.h>

#define MAXLINEA 80
#define MAXFRASE 40

int main(void)
{
  int a, b;
  char frase[MAXFRASE+1];
  char linea[MAXLINEA+1];

  printf("Dame el valor de un entero:");
```

```
    gets(linea); sscanf(linea, "%d", &a);

    printf("Introduce␣ahora␣una␣frase:");
    gets(frase);

    printf("Y␣ahora,␣dame␣el␣valor␣de␣otro␣entero:");
    gets(linea); sscanf(linea, "%d", &b);

    printf("Enteros␣leídos:␣%d,␣%d.\n", a, b);
    printf("Frase␣leída:␣%s.\n", frase);

    return 0;
}
```

En el programa hemos definido una variable auxiliar, *linea*, que es una cadena con capacidad para 80 caracteres más el terminador (puede resultar conveniente reservar más memoria para ella en según qué aplicación). Cada vez que deseamos leer un valor escalar, leemos en *linea* un texto que introduce el usuario y obtenemos el valor escalar con la función *sscanf*. Dicha función recibe, como primer argumento, la cadena en *linea*; como segundo, una cadena con marcas de formato; y como tercer parámetro, la dirección de la variable escalar en la que queremos depositar el resultado de la lectura.

Es un proceso un tanto incómodo, pero al que tenemos que acostumbrarnos... de momento.

2.2.4. Asignación y copia de cadenas

Este programa, que pretende copiar una cadena en otra, parece correcto, pero no lo es:

```
#define MAXLON 10

int main(void)
{
    char original[MAXLON+1] = "cadena";
    char copia[MAXLON+1];

    copia = original;

    return 0;
}
```

Si compilas el programa, obtendrás un error que te impedirá obtener un ejecutable. Recuerda: los identificadores de vectores estáticos se consideran punteros *inmutables* y, a fin de cuentas, las cadenas son vectores estáticos (más adelante aprenderemos a usar vectores dinámicos). Para efectuar una copia de una cadena, has de hacerlo carácter a carácter.

```
#define MAXLON 10

int main(void)
{
    char original[MAXLON+1] = "cadena";
    char copia[MAXLON+1];
    int i;

    for (i = 0; i <= MAXLON; i++)
        copia[i] = original[i];

    return 0;
}
```

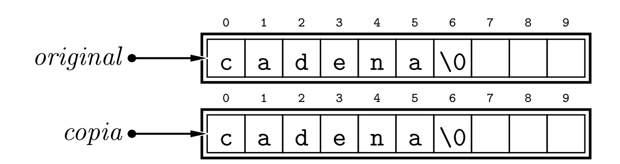
Fíjate en que el bucle recorre los 10 caracteres que realmente hay en *original* pero, de hecho, sólo necesitas copiar los caracteres que hay hasta el terminador, *incluyéndole a él*.

```
#define MAXLON 10

int main(void)
{
   char original[MAXLON+1] = "cadena";
   char copia[MAXLON+1];
   int i;

   for (i = 0; i <= MAXLON; i++) {
      copia[i] = original[i];
      if (copia[i] == '\0')
         break;
   }

   return 0;
}
```



. . . . EJERCICIOS . . . .

▶ **100** ¿Qué problema presenta esta otra versión del mismo programa?

```
#define MAXLON 10

int main(void)
{
   char original[MAXLON+1] = "cadena";
   char copia[MAXLON+1];
   int i;

   for (i = 0; i <= MAXLON; i++) {
      if (copia[i] == '\0')
         break;
      else
         copia[i] = original[i];
   }

   return 0;
}
```

. . . .

Aún podemos hacerlo «mejor»:

```
#define MAXLON 10

int main(void)
{
   char original[MAXLON+1] = "cadena";
   char copia[MAXLON+1];
   int i;

   for (i = 0; original[i] != '\0'; i++) {
```

```
        copia[i] = original[i];
    copia[i] = '\0';

    return 0;
}
```

¿Ves? La condición del **for** controla si hemos llegado al terminador o no. Como el terminador no llega a copiarse, lo añadimos tan pronto finaliza el bucle. Este tipo de bucles, aunque perfectamente legales, pueden resultar desconcertantes.

El copiado de cadenas es una acción frecuente, así que hay funciones predefinidas para ello, accesibles incluyendo la cabecera `string.h`:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
    char original[MAXLON+1] = "cadena";
    char copia[MAXLON+1];

    strcpy(copia, original); // Copia el contenido de original en copia.

    return 0;
}
```

Ten cuidado: *strcpy* (abreviatura de «string copy») no comprueba si el destino de la copia tiene capacidad suficiente para la cadena, así que puede provocar un desbordamiento. La función *strcpy* se limita a copiar carácter a carácter hasta llegar a un carácter nulo.

Tampoco está permitido asignar un literal de cadena a un vector de caracteres fuera de la zona de declaración de variables. Es decir, este programa es incorrecto:

```
#define MAXLON 10

int main(void)
{
    char a[MAXLON+1];

    a = "cadena"; // ¡Mal!

    return 0;
}
```

Si deseas asignar un literal de cadena, tendrás que hacerlo con la ayuda de *strcpy*:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
    char a[MAXLON+1];

    strcpy(a, "cadena");

    return 0;
}
```

. . . . EJERCICIOS . . . .

▶ **101** Diseña un programa que lea una cadena y copie en otra una versión encriptada. La encriptación convertirá cada letra (del alfabeto inglés) en la que le sigue en la tabla

Copias (más) seguras

Hemos dicho que *strcpy* presenta un fallo de seguridad: no comprueba si el destino es capaz de albergar todos los caracteres de la cadena original. Si quieres asegurarte de no rebasar la capacidad del vector destino puedes usar *strncpy*, una versión de *strcpy* que copia la cadena, pero con un límite al número máximo de caracteres:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
   char original[MAXLON+1] = "cadena";
   char copia[MAXLON+1];

   strncpy(copia, original, MAXLON+1); // Copia, a lo sumo, MAXLON+1 caracteres.

   return 0;
}
```

Pero tampoco *strncpy* es perfecta. Si la cadena original tiene más caracteres de los que puede almacenar la cadena destino, la copia es imperfecta: no acabará en `'\0'`. De todos modos, puedes encargarte tú mismo de terminar la cadena en el último carácter, por si acaso:

```
#include <string.h>

#define MAXLON 10

int main(void)
{
   char original[MAXLON+1] = "cadena";
   char copia[MAXLON+1];

   strncpy(copia, original, MAXLON+1);
   copia[MAXLON] = '\0';

   return 0;
}
```

ASCII (excepto en el caso de las letras «z» y «Z», que serán sustituidas por «a» y «A», respectivamente.) No uses la función *strcpy*.

▶ **102** Diseña un programa que lea una cadena que posiblemente contenga letras mayúsculas y copie en otra una versión de la misma cuyas letras sean todas minúsculas. No uses la función *strcpy*.

▶ **103** Diseña un programa que lea una cadena que posiblemente contenga letras mayúsculas y copie en otra una versión de la misma cuyas letras sean todas minúsculas. Usa la función *strcpy* para obtener un duplicado de la cadena y, después, recorre la copia para ir sustituyendo en ella las letras mayúsculas por sus correspondientes minúsculas.

. . . .

2.2.5. Longitud de una cadena

El convenio de terminar una cadena con el carácter nulo permite conocer fácilmente la longitud de una cadena:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  i = 0;
  while (a[i] != '\0')
    i++;
  printf("Longitud␣de␣la␣cadena:␣%d\n", i);

  return 0;
}
```

Calcular la longitud de una cadena es una operación frecuentemente utilizada, así que está predefinida en la biblioteca de tratamiento de cadenas. Si incluímos la cabecera `string.h`, podemos usar la función *strlen* (abreviatura de «string length»):

```
#include <stdio.h>
#include <string.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int l;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  l = strlen(a);
  printf("Longitud␣de␣la␣cadena:␣%d\n", l);

  return 0;
}
```

Has de ser consciente de qué hace *strlen*: lo mismo que hacía el primer programa, es decir, recorrer la cadena de izquierda a derecha incrementando un contador hasta llegar al terminador nulo. Esto implica que tarde tanto más cuanto más larga sea la cadena. Has de estar al tanto, pues, de la fuente de ineficiencia que puede suponer utilizar directamente *strlen* en lugares críticos como los bucles. Por ejemplo, esta función cuenta las vocales minúsculas de una cadena leída por teclado:

```
#include <stdio.h>
#include <string.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
```

El estilo C

El programa que hemos presentado para calcular la longitud de una cadena es un programa C correcto, pero no es así como un programador C expresaría esa misma idea. ¡No hace falta que el bucle incluya sentencia alguna!:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  i = 0;
  while (a[i++] != '\0') ; // Observa que no hay sentencia alguna en el while.
  printf("Longitud␣de␣la␣cadena:␣%d\n", i-1);

  return 0;
}
```

El operador de postincremento permite aumentar en uno el valor de *i* justo después de consultar el valor de *a*[*i*]. Eso sí, hemos tenido que modificar el valor mostrado como longitud, pues ahora *i* acaba valiendo uno más.

Es más, ni siquiera es necesario efectuar comparación alguna. El bucle se puede sustituir por este otro:

```
  i = 0;
  while (a[i++]) ;
```

El bucle funciona correctamente porque el valor '\0' significa «falso» cuando se interpreta como valor lógico. El bucle itera, pues, hasta llegar a un valor falso, es decir, a un terminador.

Algunos problemas con el operador de autoincremento

¿Qué esperamos que resulte de ejecutar esta sentencia?

```
  int a[5] = {0, 0, 0, 0, 0};

  i = 1;
  a[i] = i++;
```

Hay dos posibles interpretaciones:

- Se evalúa primero la parte derecha de la asignación, así que *i* pasa a valer 2 y se asigna ese valor en *a*[2].
- Se evalúa primero la asignación, con lo que se asigna el valor 1 en *a*[1] y, después, se incrementa el valor de *i*, que pasa a valer 2.

¿Qué hace C? No se sabe. La especificación del lenguaje estándar indica que el resultado está indefinido. Cada compilador elige qué hacer, así que ese tipo de sentencias pueden dar problemas de portabilidad. Conviene, pues, evitarlas.

```
  int i, contador;
```

while o for

Los bucles **while** pueden sustituirse muchas veces por bucles **for** equivalentes, bastante más compactos:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  for (i=0; a[i] != '\0'; i++) ; // Tampoco hay sentencia alguna en el for.
  printf("Longitud␣de␣la␣cadena:␣%d\n", i);

  return 0;
}
```

También aquí es superflua la comparación:

```
  for (i=0; a[i]; i++) ;
```

Todas las versiones del programa que hemos presentado son equivalentes. Escoger una u otra es cuestión de estilo.

```

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  contador = 0;
  for (i = 0; i < strlen(a); i++)
    if (a[i] == 'a' || a[i] == 'e' || a[i] == 'i' || a[i] == 'o' || a[i] == 'u')
      contador++;
  printf("Vocales␣minúsculas:␣%d\n", contador);

  return 0;
}
```

Pero tiene un problema de eficiencia. Con cada iteración del bucle **for** se llama a *strlen* y *strlen* tarda un tiempo proporcional a la longitud de la cadena. Si la cadena tiene, pongamos, 60 caracteres, se llamará a *strlen* 60 veces para efectuar la comparación, y para cada llamada, *strlen* tardará unos 60 pasos en devolver lo mismo: el valor 60. Esta nueva versión del mismo programa no presenta ese inconveniente:

```
#include <stdio.h>
#include <string.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1];
  int i, longitud, contador;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
```

```c
  longitud = strlen(cadena);
  contador = 0;
  for (i = 0; i < longitud; i++)
    if (a[i] == 'a' || a[i] == 'e' || a[i] == 'i' || a[i] == 'o' || a[i] == 'u')
      contador++;
  printf("Vocales␣minúsculas:␣%d\n", contador);

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 104 Diseña un programa que lea una cadena y la invierta.

▶ 105 Diseña un programa que lea una *palabra* y determine si es o no es palíndromo.

▶ 106 Diseña un programa que lea una *frase* y determine si es o no es palíndromo. Recuerda que los espacios en blanco y los signos de puntuación no se deben tener en cuenta a la hora de determinar si la frase es palíndromo.

▶ 107 Escribe un programa C que lea dos cadenas y muestre el índice del carácter de la primera cadena en el que empieza, por primera vez, la segunda cadena. Si la segunda cadena no está contenida en la primera, el programa nos lo hará saber.

(Ejemplo: si la primera cadena es `"un␣ejercicio␣de␣ejemplo"` y la segunda es `"eje"`, el programa mostrará el valor 3.)

▶ 108 Escribe un programa C que lea dos cadenas y muestre el índice del carácter de la primera cadena en el que empieza por *última* vez una aparición de la segunda cadena. Si la segunda cadena no está contenida en la primera, el programa nos lo hará saber.

(Ejemplo: si la primera cadena es `"un␣ejercicio␣de␣ejemplo"` y la segunda es `"eje"`, el programa mostrará el valor 16.)

▶ 109 Escribe un programa que lea una línea y haga una copia de ella eliminando los espacios en blanco que haya al principio y al final de la misma.

▶ 110 Escribe un programa que lea repetidamente líneas con el nombre completo de una persona. Para cada persona, guardará temporalmente en una cadena sus iniciales (las letras con mayúsculas) separadas por puntos y espacios en blanco y mostrará el resultado en pantalla. El programa finalizará cuando el usuario escriba una línea en blanco.

▶ 111 Diseña un programa C que lea un entero n y una cadena a y muestre por pantalla el valor (en base 10) de la cadena a si se interpreta como un número en base n. El valor de n debe estar comprendido entre 2 y 16. Si la cadena a contiene un carácter que no corresponde a un dígito en base n, notificará el error y no efectuará cálculo alguno.

Ejemplos:

- si a es `"ff"` y n es 16, se mostrará el valor 255;
- si a es `"f0"` y n es 15, se notificará un error: «`f no es un dígito en base 15`»;
- si a es `"1111"` y n es 2, se mostrará el valor 15.

▶ 112 Diseña un programa C que lea una línea y muestre por pantalla el número de palabras que hay en ella.

. . . .

2.2.6. Concatenación

Python permitía concatenar cadenas con el operador +. En C no puedes usar + para concatenar cadenas. Una posibilidad es que las concatenes tú mismo «a mano», con bucles. Este programa, por ejemplo, pide dos cadenas y concatena la segunda a la primera:

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];
  int longa, longb;
  int i;

  printf("Introduce␣un␣texto␣(máx.␣%d␣cars.):␣", MAXLON);  gets(a);
  printf("Introduce␣otro␣texto␣(máx.␣%d␣cars.):␣", MAXLON); gets(b);

  longa = strlen(a);
  longb = strlen(b);
  for (i=0; i<longb; i++)
    a[longa+i] = b[i];
  a[longa+longb] = '\0';
  printf("Concatenación␣de␣ambos:␣%s", a);

  return 0;
}
```

Pero es mejor usar la función de librería *strcat* (por «string concatenate»):

```
#include <stdio.h>
#include <string.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];

  printf("Introduce␣un␣texto␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  printf("Introduce␣otro␣texto␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(b);
  strcat(a, b);  // Equivale a la asignación Python a = a + b
  printf("Concatenación␣de␣ambos:␣%s", a);

  return 0;
}
```

Si quieres dejar el resultado de la concatenación en una variable distinta, deberás actuar en dos pasos:

```
#include <stdio.h>
#include <string.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1], c[MAXLON+1];
```

```c

  printf("Introduce un texto (máx. %d cars.): ", MAXLON);
  gets(a);
  printf("Introduce otro texto (máx. %d cars.): ", MAXLON);
  gets(b);
  strcpy(c, a);  // Ésta seguida de...
  strcat(c, b);  // ... ésta equivale a la sentencia Python c = a + b
  printf("Concatenación de ambos: %s", c);

  return 0;
}
```

Recuerda que es responsabilidad del programador asegurarse de que la cadena que recibe la concatenación dispone de capacidad suficiente para almacenar la cadena resultante.

Por cierto, el operador de repetición de cadenas que encontrábamos en Python (operador *) no está disponible en C ni hay función predefinida que lo proporcione.

> ***Un carácter no es una cadena***
>
> Un error frecuente es intentar añadir un carácter a una cadena con *strcat* o asignárselo como único carácter con *strcpy*:
>
> ```c
> char linea[10] = "cadena";
> char caracter = 's';
>
> strcat(linea, caracter); // ¡Mal!
> strcpy(linea, 'x'); // ¡Mal!
> ```
>
> Recuerda: los dos datos de *strcat* y *strcpy* han de ser *cadenas* y no es aceptable que uno de ellos sea un *carácter*.

. . . . EJERCICIOS . . . .

▶ 113 Escribe un programa C que lea el nombre y los dos apellidos de una persona en tres cadenas. A continuación, el programa formará una sóla cadena en la que aparezcan el nombre y los apellidos separados por espacios en blanco.

▶ 114 Escribe un programa C que lea un verbo regular de la primera conjugación y lo muestre por pantalla conjugado en presente de indicativo. Por ejemplo, si lee el texto `programar`, mostrará por pantalla:

```
yo programo
tú programas
él programa
nosotros programamos
vosotros programáis
ellos programan
```

. . . .

2.2.7. Comparación de cadenas

Tampoco los operadores de comparación (==, !=, <, <=, >, >=) funcionan con cadenas. Existe, no obstante, una función de `string.h` que permite paliar esta carencia de C: *strcmp* (abreviatura de «string comparison»). La función *strcmp* recibe dos cadenas, *a* y *b*, y devuelve un entero. El entero que resulta de efectuar la llamada *strcmp*(*a*, *b*) codifica el resultado de la comparación:

- es menor que cero si la cadena *a* es menor que *b*,
- es 0 si la cadena *a* es igual que *b*, y
- es mayor que cero si la cadena *a* es mayor que *b*.

Naturalmente, menor significa que va delante en orden alfabético, y mayor que va detrás.

. . . . EJERCICIOS . . . .

▶ **115** Diseña un programa C que lea dos cadenas y, si la primera es *menor o igual* que la segunda, imprima el texto «menor o igual».

▶ **116** ¿Qué valor devolverá la llamada *strcmp*("21", "112")?

▶ **117** Escribe un programa que lea dos cadenas, *a* y *b* (con capacidad para 80 caracteres), y muestre por pantalla −1 si *a* es menor que *b*, 0 si *a* es igual que *b*, y 1 si *a* es mayor que *b*. Está prohibido que utilices la función *strcmp*.

. . . .

2.2.8. Funciones útiles para manejar caracteres

No sólo `string.h` contiene funciones útiles para el tratamiento de cadenas. En `ctype.h` encontrarás unas funciones que permiten hacer cómodamente preguntas acerca de los caracteres, como si son mayúsculas, minúsculas, dígitos, etc:

- *isalnum*(*carácter*): devuelve cierto (un entero cualquiera distinto de cero) si *carácter* es una letra o dígito, y falso (el valor entero 0) en caso contrario,
- *isalpha*(*carácter*): devuelve cierto si *carácter* es una letra, y falso en caso contrario,
- *isblank*(*carácter*): devuelve cierto si *carácter* es un espacio en blanco o un tabulador,
- *isdigit*(*carácter*) devuelve cierto si *carácter* es un dígito, y falso en caso contrario,
- *isspace*(*carácter*): devuelve cierto si *carácter* es un espacio en blanco, un salto de línea, un retorno de carro, un tabulador, etc., y falso en caso contrario,
- *islower*(*carácter*): devuelve cierto si *carácter* es una letra minúscula, y falso en caso contrario,
- *isupper*(*carácter*): devuelve cierto si *carácter* es una letra mayúscula, y falso en caso contrario.

También en `ctype.h` encontrarás un par de funciones útiles para convertir caracteres de minúscula a mayúscula y viceversa:

- *toupper*(*carácter*): devuelve la mayúscula asociada a *carácter*, si la tiene; si no, devuelve el mismo carácter,
- *tolower*(*carácter*): devuelve la minúscula asociada a *carácter*, si la tiene; si no, devuelve el mismo carácter.

. . . . EJERCICIOS . . . .

▶ **118** ¿Qué problema presenta este programa?

```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char b[2] = "a";

  if (isalpha(b))
    printf("Es␣una␣letra\n");
  else
    printf("No␣es␣una␣letra\n");

  return 0;
}
```

2.2.9. Escritura en cadenas: *sprintf*

Hay una función que puede simplificar notablemente la creación de cadenas cuyo contenido se debe calcular a partir de uno o más valores: *sprintf*, disponible incluyendo la cabecera `stdio.h` (se trata, en cierto modo, de la operación complementaria de *sscanf*). La función *sprintf* se comporta como *printf*, salvo por un «detalle»: no escribe texto en pantalla, sino que lo almacena en una cadena.

Fíjate en este ejemplo:

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1] = "una";
  char b[MAXLON+1] = "cadena";
  char c[MAXLON+1];

  sprintf(c, "%s␣%s", a, b);
  printf("%s\n", c);

  return 0;
}
```

Si ejecutas el programa aparecerá lo siguiente en pantalla:

```
una cadena
```

Como puedes ver, se ha asignado a *c* el valor de *a* seguido de un espacio en blanco y de la cadena *b*. Podríamos haber conseguido el mismo efecto con llamadas a *strcpy*(*c*, *a*), *strcat*(*c*, "␣") y *strcat*(*c*, *b*), pero *sprintf* resulta más legible y no cuesta mucho aprender a usarla, pues ya sabemos usar *printf*. No olvides que tú eres responsable de que la información que se almacena en *c* quepa.

En Python hay una acción análoga al *sprintf* de C: la asignación a una variable de una cadena formada con el operador de formato. El mismo programa se podría haber escrito en Python así:

```python
# Ojo: programa Python
a = 'una'
b = 'cadena'
c = '%s␣%s' % (a, b)  # Operación análoga a sprintf en C.
print c
```

. . . . EJERCICIOS . . . .

▶ 119 ¿Qué almacena en la cadena *a* la siguiente sentencia?

```
sprintf(a, "%d-%c-%d␣%s", 1, 48, 2, "si");
```

▶ 120 Escribe un programa que pida el nombre y los dos apellidos de una persona. Cada uno de esos tres datos debe almacenarse en una variable independiente. A continuación, el programa creará y mostrará una nueva cadena con los dos apellidos y el nombre (separado de los apellidos por una coma). Por ejemplo, Juan Pérez López dará lugar a la cadena `"Pérez␣López,␣Juan"`.

. . . .

2.2.10. Un programa de ejemplo

Vamos a implementar un programa que lee por teclado una línea de texto y muestra por pantalla una cadena en la que las secuencias de blancos de la cadena original (espacios en blanco, tabuladores, etc.) se han sustituido por un sólo espacio en blanco. Si, por ejemplo, el programa lee la cadena `"una␣␣␣␣␣cadena␣␣␣␣␣con␣blancos␣␣␣"`, mostrará por pantalla la cadena «normalizada» `"una␣cadena␣con␣blancos␣"`.

normaliza.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXLON 80

int main(void)
{
  char a[MAXLON+1], b[MAXLON+1];
  int longitud, i, j;

  printf("Introduce␣una␣cadena␣(máx.␣%d␣cars.):␣", MAXLON);
  gets(a);
  longitud = strlen(a);
  b[0] = a[0];
  j = 1;
  for (i=1; i<longitud; i++)
    if (!isspace(a[i]) || (isspace(a[i]) && !isspace(a[i-1])))
      b[j++] = a[i];
  b[j] = '\0';
  printf("La␣cadena␣normalizada␣es␣%s\n", b);

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 121 Modifica `normaliza.c` para que elimine, si los hay, los blancos inicial y final de la cadena normalizada.

▶ 122 Haz un programa que lea una frase y construya una cadena que sólo contenga sus letras minúsculas o mayúsculas en el mismo orden con que aparecen en la frase.

▶ 123 Haz un programa que lea una frase y construya una cadena que sólo contenga sus letras minúsculas o mayúsculas en el mismo orden con que aparecen en la frase, pero sin repetir ninguna.

▶ 124 Lee un texto por teclado (con un máximo de 1000 caracteres) y muestra por pantalla la frecuencia de aparición de cada una de las letras del alfabeto (considera únicamente letras del alfabeto inglés), sin distinguir entre letras mayúsculas y minúsculas (una aparición de la letra `e` y otra de la letra `E` cuentan como dos ocurrencias de la letra `e`).

. . . .

2.3. Vectores multidimensionales

Podemos declarar vectores de más de una dimensión muy fácilmente:

```
int a[10][5];
float b[3][2][4];
```

En este ejemplo, *a* es una matriz de 10×5 enteros y *b* es un vector de tres dimensiones con $3 \times 2 \times 4$ números en coma flotante.

Puedes acceder a un elemento cualquiera de los vectores *a* o *b* utilizando tantos índices como dimensiones tiene el vector: `a[4][2]` y `b[1][0][3]`, por ejemplo, son elementos de *a* y *b*, respectivamente.

La inicialización de los vectores multidimensionales necesita tantos bucles anidados como dimensiones tengan éstos:

```
int main(void)
{
   int a[10][5];
   float b[3][2][4];
   int i, j, k;

   for (i=0; i<10; i++)
      for (j=0; j<5; j++)
         a[i][j] = 0;

   for (i=0; i<3; i++)
      for (j=0; j<2; j++)
         for (k=0; k<4; k++)
            b[i][j][k] = 0.0;

   return 0;
}
```

También puedes inicializar explícitamente un vector multidimensional:

```
int c[3][3] = { {1, 0, 0},
                {0, 1, 0},
                {0, 0, 1} };
```

2.3.1. Sobre la disposición de los vectores multidimensionales en memoria

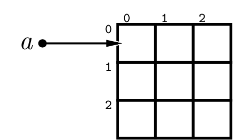
Cuando el compilador de C detecta la declaración de un vector multidimensional, reserva tantas posiciones contiguas de memoria como sea preciso para albergar todas sus celdas.

Por ejemplo, ante la declaración **int** *a*[3][3], C reserva 9 celdas de 4 bytes, es decir, 36 bytes. He aquí cómo se disponen las celdas en memoria, suponiendo que la zona de memoria asignada empieza en la dirección 1000:

996:					
1000:					a[0][0]
1004:					a[0][1]
1008:					a[0][2]
1012:					a[1][0]
1016:					a[1][1]
1020:					a[1][2]
1024:					a[2][0]
1028:					a[2][1]
1032:					a[2][2]
1036:					

Cuando accedemos a un elemento `a[i][j]`, C sabe a qué celda de memoria acceder sumando a la dirección de *a* el valor `(i*3+j)*4` (el 4 es el tamaño de un **int** y el 3 es el número de columnas).

Aun siendo conscientes de cómo representa C la memoria, nosotros trabajaremos con una representación de una matriz de 3×3 como ésta:



Como puedes ver, lo relevante es que *a* es asimilable a un puntero a la zona de memoria en la que están dispuestos los elementos de la matriz.

. . . . EJERCICIOS . . . .

▶ 125 Este programa es incorrecto. ¿Por qué? Aun siendo incorrecto, produce cierta salida por pantalla. ¿Qué muestra?

matriz_mal.c

```
#include <stdio.h>

#define TALLA 3

int main(void)
{
  int a[TALLA][TALLA];
  int i, j;

  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      a[i][j] = 10*i+j;

  for (j=0; j<TALLA*TALLA; j++)
    printf("%d\n", a[0][j]);

  return 0;
}
```

. . . .

2.3.2. Un ejemplo: cálculo matricial

Para ilustrar el manejo de vectores multidimensionales construiremos ahora un programa que lee de teclado dos matrices de números en coma flotante y muestra por pantalla

su suma y su producto. Las matrices leídas serán de 3 × 3 y se denominarán *a* y *b*. El resultado de la suma se almacenará en una matriz *s* y el del producto en otra *p*.

Aquí tienes el programa completo:

matrices.c

```
#include <stdio.h>

#define TALLA 3

int main(void)
{
  float a[TALLA][TALLA], b[TALLA][TALLA];
  float s[TALLA][TALLA], p[TALLA][TALLA];
  int i, j, k;

  /* Lectura de la matriz a */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++) {
      printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
    }

  /* Lectura de la matriz b */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++) {
      printf("Elemento (%d, %d): ", i, j); scanf("%f", &b[i][j]);
    }

  /* Cálculo de la suma */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      s[i][j] = a[i][j] + b[i][j];

  /* Cálculo del producto */
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++) {
      p[i][j] = 0.0;
      for (k=0; k<TALLA; k++)
        p[i][j] += a[i][k] * b[k][j];
    }

  /* Impresión del resultado de la suma */
  printf("Suma\n");
  for (i=0; i<TALLA; i++) {
    for (j=0; j<TALLA; j++)
      printf("%8.3f", s[i][j]);
    printf("\n");
  }

  /* Impresión del resultado del producto */
  printf("Producto\n");
  for (i=0; i<TALLA; i++) {
    for (j=0; j<TALLA; j++)
      printf("%8.3f", p[i][j]);
    printf("\n");
  }

  return 0;
}
```

Aún no sabemos definir nuestras propias funciones. En el próximo capítulo volveremos

a ver este programa y lo modificaremos para que use funciones definidas por nosotros.

EJERCICIOS

▶ 126 En una estación meteorológica registramos la temperatura (en grados centígrados) cada hora durante una semana. Almacenamos el resultado en una matriz de 7×24 (cada fila de la matriz contiene las 24 mediciones de un día). Diseña un programa que lea los datos por teclado y muestre:

- La máxima y mínima temperaturas de la semana.
- La máxima y mínima temperaturas de cada día.
- La temperatura media de la semana.
- La temperatura media de cada día.
- El número de días en los que la temperatura media fue superior a 30 grados.

▶ 127 Representamos diez ciudades con números del 0 al 9. Cuando hay carretera que une directamente a dos ciudades i y j, almacenamos su distancia en kilómetros en la celda d[i][j] de una matriz de 10×10 enteros. Si no hay carretera entre ambas ciudades, el valor almacenado en su celda de d es cero. Nos suministran un vector en el que se describe un trayecto que pasa por las 10 ciudades. Determina si se trata de un trayecto válido (las dos ciudades de todo par consecutivo están unidas por un tramo de carretera) y, en tal caso, devuelve el número de kilómetros del trayecto. Si el trayecto no es válido, indícalo con un mensaje por pantalla.

La matriz de distancias deberás inicializarla explícitamente al declararla. El vector con el recorrido de ciudades deberás leerlo de teclado.

▶ 128 Diseña un programa que lea los elementos de una matriz de 4×5 flotantes y genere un vector de talla 4 en el que cada elemento contenga el sumatorio de los elementos de cada fila. El programa debe mostrar la matriz original y el vector en este formato (evidentemente, los valores deben ser los que correspondan a lo introducido por el usuario):

```
          0       1       2       3       4        Suma
0 [ +27.33 +22.22 +10.00   +0.00 -22.22]   -> +37.33
1 [   +5.00   +0.00   -1.50   +2.50 +10.00]   -> +16.00
2 [   +3.45   +2.33   -4.56 +12.56 +12.01]   -> +25.79
3 [   +1.02   +2.22 +12.70 +34.00 +12.00]   -> +61.94
```

El programa que hemos presentado adolece de un serio inconveniente si nuestro objetivo era construir un programa «general» para multiplicar matrices: sólo puede trabajar con matrices de TALLA × TALLA, o sea, de 3×3. ¿Y si quisiéramos trabajar con matrices de tamaños arbitrarios? El primer problema al que nos enfrentaríamos es el de que las matrices han de tener una talla máxima: no podemos, con lo que sabemos por ahora, reservar un espacio de memoria para las matrices que dependa de datos que nos suministra el usuario en tiempo de ejecución. Usaremos, pues, una constante `MAXTALLA` con un valor razonablemente grande: pongamos 10. Ello permitirá trabajar con matrices con un número de filas y columnas menor o igual que 10, aunque será a costa de malgastar memoria.

matrices.c

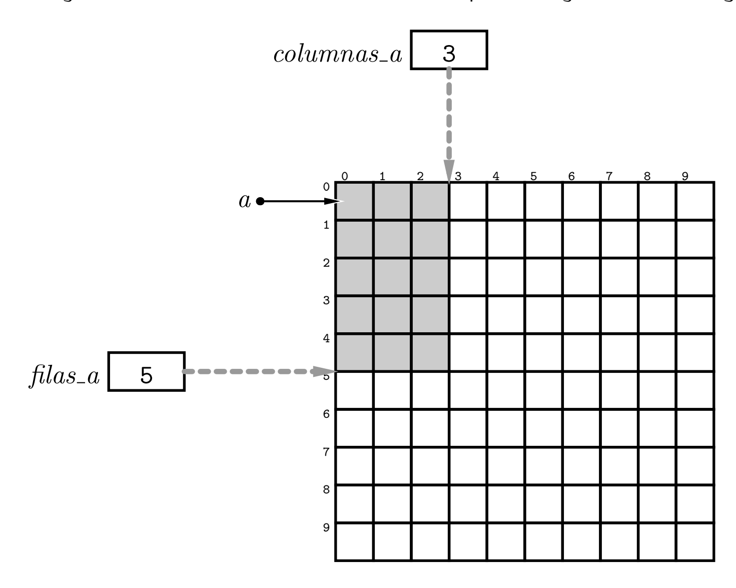
```
#include <stdio.h>

#define MAXTALLA 10

```

```
int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
  ...
```

El número de filas y columnas de *a* se pedirá al usuario y se almacenará en sendas variables: *filas_a* y *columnas_a*. Este gráfico ilustra su papel: la matriz *a* es de 10×10, pero sólo usamos una parte de ella (la zona sombreada) y podemos determinar qué zona es porque *filas_a* y *columnas_a* nos señalan hasta qué fila y columna llega la zona útil:



Lo mismo se aplicará al número de filas y columnas de b. Te mostramos el programa hasta el punto en que leemos la matriz *a*:

matrices.c

```
#include <stdio.h>

#define MAXTALLA 10

int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
  int filas_a, columnas_a, filas_b, columnas_b;
  int i, j, k;

  /* Lectura de la matriz a */
  printf("Filas␣de␣a␣␣␣:␣"); scanf("%d", &filas_a);
  printf("Columnas␣de␣a:␣"); scanf("%d", &columnas_a);

  for (i=0; i<filas_a; i++)
    for (j=0; j<columnas_a; j++) {
      printf("Elemento␣(%d,␣%d):␣", i, j); scanf("%f", &a[i][j]);
    }
  ...
```

(Encárgate tú mismo de la lectura de b.)

La suma sólo es factible si `filas_a` es igual a `filas_b` y `columnas_a` es igual a `columnas_b`.

matrices.c

```c
#include <stdio.h>

#define MAXTALLA 10

int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
  int filas_a, columnas_a, filas_b, columnas_b;
  int filas_s, columnas_s;
  int i, j, k;

  /* Lectura de la matriz a */
  printf("Filas de a   : "); scanf("%d", &filas_a);
  printf("Columnas de a: "); scanf("%d", &columnas_a);
  for (i=0; i<filas_a; i++)
    for (j=0; j<columnas_a; j++) {
      printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
    }

  /* Lectura de la matriz b */
  ...

  /* Cálculo de la suma */
  if (filas_a == filas_b && columnas_a == columnas_b) {
    filas_s = filas_a;
    columnas_s = columnas_a;
    for (i=0; i<filas_s; i++)
      for (j=0; j<filas_s; j++)
        s[i][j] = a[i][j] + b[i][j];
  }

  /* Impresión del resultado de la suma */
  if (filas_a == filas_b && columnas_a == columnas_b) {
    printf("Suma\n");
    for (i=0; i<filas_s; i++) {
      for (j=0; j<columnas_s; j++)
        printf("%8.3f", s[i][j]);
      printf("\n");
    }
  }
  else
    printf("Matrices no compatibles para la suma.\n");

  ...
```

Recuerda que una matriz de $n \times m$ elementos se puede multiplicar por otra de $n' \times m'$ elementos sólo si m es igual a n' (o sea, el número de columnas de la primera es igual al de filas de la segunda) y que la matriz resultante es de dimensión $n \times m'$.

matrices_1.c — **matrices.c**

```c
#include <stdio.h>

#define MAXTALLA 10

int main(void)
{
  float a[MAXTALLA][MAXTALLA], b[MAXTALLA][MAXTALLA];
  float s[MAXTALLA][MAXTALLA], p[MAXTALLA][MAXTALLA];
```

```
  int filas_a, columnas_a, filas_b, columnas_b;
  int filas_s, columnas_s, filas_p, columnas_p;
  int i, j, k;

  /* Lectura de la matriz a */
  printf("Filas de a   : "); scanf("%d", &filas_a);
  printf("Columnas de a: "); scanf("%d", &columnas_a);
  for (i=0; i<filas_a; i++)
    for (j=0; j<columnas_a; j++) {
      printf("Elemento (%d, %d): ", i, j); scanf("%f", &a[i][j]);
    }

  /* Lectura de la matriz b */
  printf("Filas de b   : "); scanf("%d", &filas_b);
  printf("Columnas de b: "); scanf("%d", &columnas_b);
  for (i=0; i<filas_b; i++)
    for (j=0; j<columnas_b; j++) {
      printf("Elemento (%d, %d): ", i, j); scanf("%f", &b[i][j]);
    }

  /* Cálculo de la suma */
  if (filas_a == filas_b && columnas_a == columnas_b) {
    filas_s = filas_a;
    columnas_s = columnas_a;
    for (i=0; i<filas_s; i++)
      for (j=0; j<filas_s; j++)
        s[i][j] = a[i][j] + b[i][j];
  }

  /* Cálculo del producto */
  if (columnas_a == filas_b) {
    filas_p = filas_a;
    columnas_p = columnas_b;
    for (i=0; i<filas_p; i++)
      for (j=0; j<columnas_p; j++) {
        p[i][j] = 0.0;
        for (k=0; k<columnas_a; k++)
          p[i][j] += a[i][k] * b[k][j];
      }
  }

  /* Impresión del resultado de la suma */
  if (filas_a == filas_b && columnas_a == columnas_b) {
    printf("Suma\n");
    for (i=0; i<filas_s; i++) {
      for (j=0; j<columnas_s; j++)
        printf("%8.3f", s[i][j]);
      printf("\n");
    }
  }
  else
    printf("Matrices no compatibles para la suma.\n");

  /* Impresión del resultado del producto */
  if (columnas_a == filas_b) {
    printf("Producto\n");
    for (i=0; i<filas_p; i++) {
      for (j=0; j<columnas_p; j++)
        printf("%8.3f", p[i][j]);
```

```
        printf("\n");
      }
    }
    else
      printf("Matrices no compatibles para el producto.\n");

    return 0;
}
```

. . . . EJERCICIOS . . . .

▶ **129** Extiende el programa de calculadora matricial para efectuar las siguientes operaciones:

- Producto de una matriz por un escalar. (La matriz resultante tiene la misma dimensión que la original y cada elemento se obtiene multiplicando el escalar por la celda correspondiente de la matriz original.)
- Transpuesta de una matriz. (La transpuesta de una matriz de $n \times m$ es una matriz de $m \times n$ en la que el elemento de la fila i y columna j tiene el mismo valor que el que ocupa la celda de la fila j y columna i en la matriz original.)

▶ **130** Una matriz tiene un valle si el valor de una de sus celdas es menor que el de cualquiera de sus 8 celdas vecinas. Diseña un programa que lea una matriz (el usuario te indicará de cuántas filas y columnas) y nos diga si la matriz tiene un valle o no. En caso afirmativo, nos mostrará en pantalla las coordenadas de *todos* los valles, sus valores y el de sus celdas vecinas.

La matriz debe tener un número de filas y columnas mayor o igual que 3 y menor o igual que 10. Las casillas que no tienen 8 vecinos no se consideran candidatas a ser valle (pues no tienen 8 vecinos).

Aquí tienes un ejemplo de la salida esperada para esta matriz de 4×5:

$$\begin{pmatrix} 1 & 2 & 9 & 5 & 5 \\ 3 & 2 & 9 & 4 & 5 \\ 6 & 1 & 8 & 7 & 6 \\ 6 & 3 & 8 & 0 & 9 \end{pmatrix}$$

```
Valle en fila 2 columna 4:
9 5 5
9 4 5
8 7 6
Valle en fila 3 columna 2:
3 2 9
6 1 8
6 3 8
```

(Observa que al usuario se le muestran filas y columnas numeradas desde 1, y no desde 0.)

▶ **131** Modifica el programa del ejercicio anterior para que considere candidata a valle a cualquier celda de la matriz. Si una celda tiene menos de 8 vecinos, se considera que la celda es valle si su valor es menor que el de todos ellos.

Para la misma matriz del ejemplo del ejercicio anterior se obtendría esta salida:

```
Valle en fila 1 columna 1:
x x x
x 1 2
x 3 2
```

```
Valle en fila 2 columna 4:
9 5 5
9 4 5
8 7 6
Valle en fila 3 columna 2:
3 2 9
6 1 8
6 3 8
Valle en fila 4 columna 4:
8 7 6
8 0 9
x x x
```

2.3.3. Vectores de cadenas, matrices de caracteres

Por lo dicho hasta el momento, está claro que un vector de cadenas es una matriz de caracteres. Este fragmento de programa, por ejemplo, declara un vector de 10 cadenas cuya longitud es menor o igual que 80:

```
#define MAXLON 80

char v[10][MAXLON+1];
```

Cada fila de la matriz es una cadena y, como tal, debe terminar en un carácter nulo.

Este fragmento declara e inicializa un vector de tres cadenas:

```
#define MAXLON 80

char v[3][MAXLON+1] = {"una",
                       "dos",
                       "tres" };
```

Puedes leer individualmente cada cadena por teclado:

matriz_cadenas.c

```
#include <stdio.h>

#define MAXLON 80

int main(void)
{
  char v[3][MAXLON+1];
  int i;

  for (i=0; i<3; i++) {
    printf("Introduzca␣cadena:␣");
    gets(v[i]);
    printf("Cadena␣leída:␣%s\n", v[i]);
  }

  return 0;
}
```

Vamos a desarrollar un programa útil que hace uso de un vector de caracteres: un pequeño corrector ortográfico para inglés. El programa dispondrá de una lista de palabras en inglés (que encontrarás en la página web de la asignatura, en el fichero `ingles.h`), solicitará al usuario que introduzca por teclado un texto en inglés y le informará de qué palabras considera erróneas por no estar incluídas en su diccionario. Aquí tienes un ejemplo de uso del programa:

```
Introduce una frase: does this sentence contiene only correct words, eh? ↲
palabra no encontrada: contiene
palabra no encontrada: eh
```

El fichero `ingles.h` es una cabecera de la que te mostramos ahora las primeras y últimas líneas:

ingles.h **ingles.h**

```
#define DICCPALS 45378
#define MAXLONPAL 28
char diccionario[DICCPALS][MAXLONPAL+1] = {
"aarhus",
"aaron",
"ababa",
"aback",
"abaft",
"abandon",
"abandoned",
"abandoning",
"abandonment",
⋮
"zorn",
"zoroaster",
"zoroastrian",
"zulu",
"zulus",
"zurich"
};
```

La variable *diccionario* es un vector de cadenas (o una matriz de caracteres, según lo veas) donde cada elemento es una palabra inglesa en minúsculas. La constante `DICCPALS` nos indica el número de palabras que contiene el diccionario y `MAXLONPAL` es la longitud de la palabra más larga (28 bytes), por lo que reservamos espacio para `MAXLONPAL+1` caracteres (29 bytes: 28 más el correspondiente al terminador nulo).

Las primeras líneas de nuestro programa son éstas:

corrector.c

```
#include <stdio.h>
#include "ingles.h"
```

Fíjate en que incluímos el fichero `ingles.h` encerrando su nombre entre comillas dobles, y no entre < y >. Hemos de hacerlo así porque `ingles.h` es una cabecera nuestra y no reside en los directorios estándar del sistema (más sobre esto en el siguiente capítulo).

El programa empieza solicitando una cadena con *gets*. A continuación, la dividirá en un nuevo vector de palabras. Supondremos que una frase no contiene más de 100 palabras y que una palabra es una secuencia cualquiera de letras. Si el usuario introduce más de 100 palabras, le advertiremos de que el programa sólo corrige las 100 primeras. Una vez formada la lista de palabras de la frase, el programa buscará cada una de ellas en el diccionario. Las que no estén, se mostrarán en pantalla precedidas del mensaje: `palabra no encontrada`. Vamos allá: empezaremos por la lectura de la frase y su descomposición en una lista de palabras.

corrector_1.c **corrector.c**

```
#include <stdio.h>
#include "ingles.h"
#include <string.h>
#include <ctype.h>
```

```
5
6  #define MAXLONFRASE 1000
7  #define MAXPALSFRASE 100
8  #define MAXLONPALFRASE 100
9
10 int main(void)
11 {
12   char frase[MAXLONFRASE+1];
13   char palabra[MAXPALSFRASE][MAXLONPALFRASE+1];
14   int palabras; // Número de palabras en la frase
15   int lonfrase, i, j;
16
17   /* Lectura de la frase */
18   printf("Introduce␣una␣frase:␣");
19   gets(frase);
20
21   lonfrase = strlen(frase);
22
23   /* Descomposición en un vector de palabras */
24   i = 0;
25   while (i<lonfrase && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
26
27   palabras = 0;
28   while (i<lonfrase) { // Recorrer todos los caracteres
29
30     // Avanzar mientras vemos caracteres e ir formando la palabra palabra[palabras].
31     j = 0;
32     while (i<lonfrase && isalpha(frase[i])) palabra[palabras][j++] = frase[i++];
33     palabra[palabras][j] = '\0';  // El terminador es responsabilidad nuestra.
34
35     // Incrementar el contador de palabras.
36     palabras++;
37     if (palabras == MAXPALSFRASE) // Y finalizar si ya no caben más palabras
38       break;
39
40     // Saltarse las no-letras que separan esta palabra de la siguiente (si las hay).
41     while (i<lonfrase && !isalpha(frase[i])) i++;
42   }
43
44   /* Comprobación de posibles errores */
45   for (i=0; i<palabras; i++)
46     printf("%s\n", palabra[i]);
47
48   return 0;
49 }
```

¡Buf! Complicado, ¿no? ¡Ya estamos echando en falta el método *split* de Python! No nos viene mal probar si nuestro código funciona mostrando las palabras que ha encontrado en la frase. Por eso hemos añadido las líneas 44–46. Una vez hayas ejecutado el programa y comprobado que funciona correctamente hasta este punto, comenta el bucle que muestra las palabras:

```
44   /* Comprobación de posibles errores */
45   // for (i=0; i<palabras; i++)
46   // printf("%s\n", palabra[i]);
```

. . . . EJERCICIOS . . . .

▶ **132** Un programador, al copiar el programa, ha sustituido la línea que reza así:

```
  while (i<lonfrase && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
```

por esta otra:

```
while (frase[i] != '\0' && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
```

¿Es correcto el programa resultante? ¿Por qué?

▶ 133 Un programador, al copiar el programa, ha sustituido la línea que reza así:

```
while (i<lonfrase && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
```

por esta otra:

```
while (frase[i] && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
```

¿Es correcto el programa resultante? ¿Por qué?

▶ 134 Un programador, al copiar el programa, ha sustituido la línea que reza así:

```
while (i<lonfrase && isalpha(frase[i])) palabra[palabras][j++] = frase[i++];
```

por esta otra:

```
while (isalpha(frase[i])) palabra[palabras][j++] = frase[i++];
```

¿Es correcto el programa resultante? ¿Por qué?

▶ 135 Un programador, al copiar el programa, ha sustituido la línea que reza así:

```
while (i<lonfrase && !isalpha(frase[i])) i++; // Saltarse las no-letras iniciales.
```

por esta otra:

```
while (!isalpha(frase[i])) palabra[palabras][j++] = frase[i++];
```

¿Es correcto el programa resultante? ¿Por qué?

. . . .

Sigamos. Nos queda la búsqueda de cada palabra en el diccionario. Una primera idea consiste en buscar cada palabra de la frase recorriendo el diccionario desde la primera hasta la última entrada:

corrector_2.c **corrector.c**

```
  ⋮

  /* ¿Están todas las palabras en el diccionario? */
  for (i=0; i<palabras; i++) {
    encontrada = 0;
    for (j=0; j<DICCPALS; j++)
      if (strcmp(palabra[i],diccionario[j]) == 0) { // ¿Es palabra[i] igual que diccionario[j]?
        encontrada = 1;
        break;
      }
    if (!encontrada)
      printf("palabra no encontrada: %s\n", palabra[i]);
  }
  return 0;
}
```

Ten en cuenta lo que hace *strcmp*: recorre las dos cadenas hasta encontrar alguna diferencia entre ellas o concluir que son idénticas. Es, por tanto, una operación bastante costosa en tiempo. ¿Podemos reducir el número de comparaciones? ¡Claro! Como el diccionario está ordenado alfabéticamente, podemos abortar el recorrido cuando llegamos a una voz del diccionario posterior (según el orden alfabético) a la que buscamos:

corrector_3.c

corrector.c

```
⋮
    /* ¿Están todas las palabras en el diccionario? */
    for (i=0; i<palabras; i++) {
        encontrada = 0;
        for (j=0; j<DICCPALS; j++)
            // ¿Es palabra[i] igual que diccionario[j]?
            if (strcmp(palabra[i],diccionario[j]) == 0) {
                encontrada = 1;
                break;
            }
            else if (strcmp(palabra[i], diccionario[j]) < 0) // ¿palabra[i] < diccionario[j]?
                break;
        if (!encontrada)
            printf("palabra␣no␣encontrada:␣%s\n", palabra[i]);
    }
    return 0;
}
```

Con esta mejora hemos *intentado* reducir a la mitad el número de comparaciones con cadenas del diccionario, pero no hemos logrado nuestro objetivo: ¡aunque, en promedio, efectuamos comparaciones con la mitad de las palabras del diccionario, estamos llamando dos veces a `strcmp`! Es mejor almacenar el resultado de una sola llamada a `strcmp` en una variable:

corrector_4.c

corrector.c

```
⋮
    /* ¿Están todas las palabras en el diccionario? */
    for (i=0; i<palabras; i++) {
        encontrada = 0;
        for (j=0; j<DICCPALS; j++) {
            comparacion = strcmp(palabra[i], diccionario[j]);
            if (comparacion == 0) {      // ¿Es palabra[i] igual que diccionario[j]?
                encontrada = 1;      break;
            }
            else if (comparacion < 0) // ¿Es palabra[i] menor que diccionario[j]?
                break;
        }
        if (!encontrada)
            printf("palabra␣no␣encontrada:␣%s\n", palabra[i]);
    }
    return 0;
}
```

(Recuerda declarar *comparacion* como variable de tipo entero.)

El diccionario tiene 45378 palabras. En promedio efectuamos, pues, 22689 comparaciones por cada palabra de la frase. Mmmm. Aún podemos hacerlo mejor. Si la lista está ordenada, podemos efectuar una búsqueda dicotómica. La búsqueda dicotómica efectúa un número de comparaciones reducidísimo: ¡bastan 16 comparaciones para decidir si una palabra cualquiera está o no en el diccionario!

corrector.c

```
⋮
    /* ¿Están todas las palabras en el diccionario? */
```

```
  for (i=0; i<palabras; i++) {
    encontrada = 0;
    izquierda = 0;
    derecha = DICCPALS;

    while (izquierda < derecha) {
      j = (izquierda + derecha) / 2;
      comparacion = strcmp(palabra[i], diccionario[j]);
      if (comparacion < 0)
        derecha = j;
      else if (comparacion > 0)
        izquierda = j+1;
      else {
        encontrada = 1;
        break;
      }
    }

    if (!encontrada)
      printf("palabra␣no␣encontrada:␣%s\n", palabra[i]);
  }

  return 0;
}
```

(Debes declarar *derecha* e *izquierda* como enteros.)

. . . . EJERCICIOS . . . .

▶ 136 Escribe un programa C que lea un texto (de longitud menor que 1000) y obtenga un vector de cadenas en el que cada elemento es una palabra distinta del texto (con un máximo de 500 palabras). Muestra el contenido del vector por pantalla.

▶ 137 Modifica el programa del ejercicio anterior para que el vector de palabras se muestre en pantalla ordenado alfabéticamente. Deberás utilizar el método de la burbuja para ordenar el vector.

▶ 138 Representamos la baraja de cartas con un vector de cadenas. Los palos son "*oros*", "copas", "espadas" y "bastos". Las cartas con números entre 2 y 9 se describen con el texto "número␣de␣palo" (ejemplo: "2␣de␣oros", "6␣de␣copas"). Los ases se describen con la cadena "as␣de␣palo", las sotas con "sota␣de␣palo", los caballos con "caballo␣de␣palo" y los reyes con "rey␣de␣palo".

Escribe un programa que genere la descripción de las 40 cartas de la baraja. Usa bucles siempre que puedas y compón las diferentes partes de cada descripción con *strcat* o *sprintf*. A continuación, baraja las cartas utilizando para ello el generador de números aleatorios y muestra el resultado por pantalla.

▶ 139 Diseña un programa de ayuda al diagnóstico de enfermedades. En nuestra base de datos hemos registrado 10 enfermedades y 10 síntomas:

```
char enfermedades[10][20] = { "gripe", "indigestión", "catarro", ... };
char sintomas[10][20] = { "fiebre", "tos", "dolor␣de␣cabeza", ... };
```

Almacenamos en una matriz de 10×10 valores booleanos (1 o 0) los síntomas que presenta cada enfermedad:

```
char sintomatologia[10][10] = {{ 1, 0, 1, ... },
                               { 0, 0, 0, ... },
                               ...
                              };
```

La celda *sintomatologia*[*i*][*j*] vale 1 si la enfermedad *i* presenta el síntoma *j*, y 0 en caso contrario.

Diseña un programa que pregunte al paciente si sufre cada uno de los 10 síntomas y, en función de las respuestas dadas, determine la enfermedad que padece. Si la descripción de sus síntomas no coincide exactamente con la de alguna de las enfermedades, el sistema indicará que no se puede emitir un diagnóstico fiable.

▶ 140 Modifica el programa anterior para que, cuando no hay coincidencia absoluta de síntomas, muestre las tres enfermedades con sintomatología más parecida. Si, por ejemplo, una enfermedad presenta 9 coincidencias con la sintomatología del paciente, el sistema mostrará el nombre de la enfermedad y el porcentaje de confianza del diagnóstico (90%).

▶ 141 Vamos a implementar un programa que nos ayude a traducir texto a código Morse. Aquí tienes una tabla con el código Morse:

A .-	B -...	C -.-.	D -..	E .	F ..-.	G -.	H	I ..	J .--	K -.-	L .-..
M -	N -.	O --	P .-.	Q -.-	R .-.	S ...	T -	U ..-	V ...-	W .-	X -..-
Y -.-	Z -..	0 ---	1 .--	2 ..--	3 ...-	4 ....-	5	6 -....	7 -...	8 --..	9 --.

El programa leerá una línea y mostrará por pantalla su traducción a código Morse. Ten en cuenta que las letras se deben separar por pausas (un espacio blanco) y las palabras por pausas largas (tres espacios blancos). Los acentos no se tendrán en cuenta al efectuar la traducción (la letra `Á`, por ejemplo, se representará con `.-`) y la letra `'Ñ'` se mostrará como una `'N'`. Los signos que no aparecen en la tabla (comas, admiraciones, etc.) no se traducirán, excepción hecha del punto, que se traduce por la palabra `STOP`. Te conviene pasar la cadena a mayúsculas (o efectuar esta transformación sobre la marcha), pues la tabla Morse sólo recoge las letras mayúsculas y los dígitos.

Por ejemplo, la cadena `"Hola,␣mundo."` se traducirá por

```
.... -- .-.. .-   - ..- -. -.. --   ... - -- .-.
```

Debes usar un vector de cadenas para representar la tabla de traducción a Morse. El código Morse de la letra `'A'`, por ejemplo, estará accesible como una cadena en *morse*`['A']`.

(Tal vez te sorprenda la notación *morse*`['A']`. Recuerda que `'A'` *es* el número 65, pues el carácter `'A'` tiene ese valor ASCII. Así pues, *morse*`['A']` y *morse*`[65]` son lo mismo. Por cierto: el vector de cadenas *morse* sólo tendrá códigos para las letras mayúsculas y los dígitos; recuerda inicializar el resto de componentes con la cadena vacía.)

▶ 142 Escribe un programa que lea un texto escrito en código Morse y lo traduzca al código alfabético.

Si, por ejemplo, el programa lee por teclado esta cadena:

```
"....␣--␣.-..␣.-␣-␣..-␣-.␣-..␣--␣...␣-␣--␣.-."
```

mostrará en pantalla el texto `HOLAMUNDOSTOP`.

. . . .

2.4. Registros

Los vectores permiten agrupar varios elementos de un mismo tipo. Cada elemento de un vector es accesible a través de un índice.

En ocasiones necesitarás agrupar datos de diferentes tipos y/o preferirás acceder a diferentes elementos de un grupo de datos a través de un identificador, no de un índice. Los

registros son agrupaciones heterogéneas de datos cuyos elementos (denominados *campos*) son accesibles mediante identificadores. Ya hemos estudiado registros en Python, así que el concepto y su utilidad han de resultarte familiares.

Veamos ahora un diseño típico de registro. Supongamos que deseamos mantener los siguientes datos de una persona:

- su nombre (con un máximo de 40 caracteres),
- su edad (un entero),
- su DNI (una cadena de 9 caracteres).

Podemos definir un registro «persona» antes de la aparición de *main*:

```
#define MAXNOM 40
#define LONDNI 9

struct Persona {
  char nombre[MAXNOM+1];
  int  edad;
  char dni[LONDNI+1];
}; // <- Fíjate en el punto y coma: es fácil olvidarse de ponerlo.
```

La definición de un registro introduce un nuevo tipo de datos en nuestro programa. En el ejemplo hemos definido el tipo **struct** *Persona* (la palabra **struct** forma parte del nombre del tipo). Ahora puedes declarar variables de tipo **struct** *Persona* así:

```
struct Persona pepe, juan, ana;
```

En tu programa puedes acceder a cada uno de los campos de una variable de tipo **struct** separando con un punto el identificador de la variable del correspondiente identificador del campo. Por ejemplo, *pepe.edad* es la edad de Pepe (un entero que ocupa cuatro bytes), *juan.nombre* es el nombre de Juan (una cadena), y *ana.dni*[8] es la letra del DNI de Ana (un carácter).

Cada variable de tipo **struct** *Persona* ocupa, *en principio*, 55 bytes: 41 por el nombre, 4 por la edad y 10 por el DNI. (Si quieres saber por qué hemos resaltado lo de «en principio», lee el cuadro «Alineamientos».)

Este programa ilustra cómo acceder a los campos de un registro leyendo por teclado sus valores y mostrando por pantalla diferentes informaciones almacenadas en él:

registro.c

```
#include <stdio.h>
#include <string.h>

#define MAXNOM 40
#define LONDNI 9

struct Persona {
  char nombre[MAXNOM+1];
  int edad;
  char dni[LONDNI+1];
};

int main(void)
{
  struct Persona ejemplo;
  char linea[81];
  int i, longitud;

```

Alineamientos

El operador **sizeof** devuelve el tamaño en bytes de un tipo o variable. Analiza este programa:

alineamiento.c

```
#include <stdio.h>

struct Registro {
  char a;
  int b;
};

int main(void)
{
  printf("Ocupación:␣%d␣bytes\n", sizeof(struct Registro));
  return 0;
}
```

Parece que vaya a mostrar en pantalla el mensaje «`Ocupación: 5 bytes`», pues un **char** ocupa 1 byte y un **int** ocupa 4. Pero no es así:

```
Ocupación: 8 bytes
```

La razón de que ocupe más de lo previsto es la eficiencia. Los ordenadores con arquitectura de 32 bits agrupan la información en bloques de 4 bytes. Cada uno de esos bloques se denomina «palabra». Cada acceso a memoria permite traer al procesador los 4 bytes de una palabra. Si un dato está a caballo entre dos palabras, requiere dos accesos a memoria, afectando seriamente a la eficiencia del programa. El compilador trata de generar un programa eficiente y da prioridad a la velocidad de ejecución frente al consumo de memoria. En nuestro caso, esta prioridad se ha traducido en que el segundo campo se almacene en una palabra completa, aunque ello suponga desperdiciar 3 bytes en el primero de los campos.

```
  printf("Nombre:␣"); gets(ejemplo.nombre);
  printf("Edad␣␣:␣"); gets(linea); sscanf(linea, "%d", &ejemplo.edad);
  printf("DNI␣␣␣:␣"); gets(ejemplo.dni);

  printf("Nombre␣leído:␣%s\n", ejemplo.nombre);
  printf("Edad␣leída␣␣:␣%d\n", ejemplo.edad);
  printf("DNI␣leído␣␣␣:␣%s\n", ejemplo.dni);

  printf("Iniciales␣del␣nombre:␣");
  longitud = strlen(ejemplo.nombre);
  for (i=0; i<longitud; i++)
    if (ejemplo.nombre[i] >= 'A' && ejemplo.nombre[i] <= 'Z')
      printf("%c", ejemplo.nombre[i]);
  printf("\n");

  printf("Letra␣del␣DNI:␣");
  longitud = strlen(ejemplo.dni);
  if (ejemplo.dni[longitud-1] < 'A' || ejemplo.dni[longitud-1] > 'Z')
    printf("No␣tiene␣letra.\n");
  else
    printf("%c\n", ejemplo.dni[longitud-1]);

  return 0;
}
```

Los registros pueden copiarse íntegramente sin mayor problema. Este programa, por ejemplo, copia el contenido de un registro en otro y pasa a minúsculas el nombre de la copia:

copia_registro.c

copia_registro.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXNOM 40
#define LONDNI 9

struct Persona {
  char nombre[MAXNOM+1];
  int  edad;
  char dni[LONDNI+1];
};

int main(void)
{
  struct Persona una, copia;
  char linea[81];
  int i, longitud;

  printf("Nombre: "); gets(una.nombre);
  printf("Edad   : "); gets(linea); sscanf(linea, "%d", &una.edad);
  printf("DNI    : "); gets(una.dni);

  copia = una; // Copia

  longitud = strlen(copia.nombre);
  for (i=0; i<longitud; i++)
    copia.nombre[i] = tolower(copia.nombre[i]);

  printf("Nombre leído: %s\n", una.nombre);
  printf("Edad leída  : %d\n", una.edad);
  printf("DNI leído   : %s\n", una.dni);

  printf("Nombre copia: %s\n", copia.nombre);
  printf("Edad copia  : %d\n", copia.edad);
  printf("DNI copia   : %s\n", copia.dni);

  return 0;
}
```

Observa que la copia se efectúa incluso cuando los elementos del registro son vectores. O sea, copiar vectores con una mera asignación está prohibido, pero copiar registros es posible. Un poco incoherente, ¿no?

Por otra parte, no puedes comparar registros. Este programa, por ejemplo, efectúa una copia de un registro en otro para, a continuación, intentar decirnos si ambos son iguales o no:

compara_registros_mal.c

```
#include <stdio.h>

#define MAXNOM 40
#define LONDNI 9

struct Persona {
```

```c
7     char nombre[MAXNOM+1];
8     int  edad;
9     char dni[LONDNI+1];
10 };
11
12 int main(void)
13 {
14    struct Persona una, copia;
15    char linea[81];
16    int i, longitud;
17
18    printf("Nombre:␣"); gets(una.nombre);
19    printf("Edad␣␣:␣"); gets(linea); sscanf(linea, "%d", &una.edad);
20    printf("DNI␣␣␣:␣"); gets(una.dni);
21
22    copia = una; // Copia
23
24    if (copia == una) // Comparación ilegal.
25       printf("Son␣iguales\n");
26    else
27       printf("No␣son␣iguales\n");
28
29    return 0;
30 }
```

Pero ni siquiera es posible compilarlo. La línea 24 contiene un error que el compilador señala como «`invalid operands to binary ==`», o sea, «operandos inválidos para la operación binaria `==`». Entonces, ¿cómo podemos decidir si dos registros son iguales? Comprobando la igualdad de cada uno de los campos de un registro con el correspondiente campo del otro:

compara_registros.c   **compara_registros.c**

```c
#include <stdio.h>

#define MAXNOM 40
#define LONDNI 9

struct Persona {
   char nombre[MAXNOM+1];
   int  edad;
   char dni[LONDNI+1];
};

int main(void)
{
   struct Persona una, copia;
   char linea[81];
   int i, longitud;

   printf("Nombre:␣"); gets(una.nombre);
   printf("Edad␣␣:␣"); gets(linea); scanf(linea, "%d", &una.edad);
   printf("DNI␣␣␣:␣"); gets(una.dni);

   copia = una; // Copia

   if (strcmp(copia.nombre, una.nombre)==0 && copia.edad==una.edad
         && strcmp(copia.dni, una.dni)==0)
      printf("Son␣iguales\n");
   else
      printf("No␣son␣iguales\n");
```

```
	return 0;
}
```

Una razón para no comparar

Si C sabe copiar una estructura «bit a bit», ¿por qué no sabe compararlas «bit a bit»? El problema estriba en construcciones como las cadenas que son campos de un registro. Considera esta definición:

```
struct Persona {
	char nombre[10];
	char apellido[10];
};
```

Cada dato de tipo **struct** *Persona* ocupa 20 bytes. Si una persona *a* tiene su campo *a.nombre* con valor "Pepe", sólo los cinco primeros bytes de su nombre tienen un valor bien definido. Los cinco siguientes pueden tener cualquier valor aleatorio. Otro registro *b* cuyo campo *b.nombre* también valga "Pepe" (y tenga idéntico apellido) puede tener valores diferentes en su segundo grupo de cinco bytes. Una comparación «bit a bit» nos diría que los registros son diferentes.

La asignación no entraña este tipo de problema, pues la copia es «bit a bit». Como mucho, resulta algo ineficiente, pues copiará hasta los bytes de valor indefinido.

Una forma de inicialización

C permite inicializar registros de diferentes modos, algunos bastante interesantes desde el punto de vista de la legibilidad. Este programa, por ejemplo, define un **struct** y crea e inicializa de diferentes formas, pero con el mismo valor, varias variables de este tipo:

```
struct Algo {
	int x;
	char nombre[10];
	float y;
};

...

	struct Algo a = { 1, "Pepe", 2.0 };
	struct Algo b = { .x = 1, .nombre = "Pepe", .y = 2.0 };
	struct Algo c = { .nombre = "Pepe", .y = 2.0, .x = 1};
	struct Algo d;
	...
	d.x = 1;
	strcpy(d.nombre, "Pepe");
	d.y = 2.0;
```

2.4.1. Un ejemplo: registros para almacenar vectores de talla variable (pero acotada)

Los vectores estáticos tienen una talla fija. Cuando necesitamos un vector cuya talla varía o no se conoce hasta iniciada la ejecución del programa usamos un truco: definimos un vector cuya talla sea suficientemente grande para la tarea que vamos a abordar y

mantenemos la «talla real» en una variable. Lo hemos hecho con el programa que calcula algunas estadísticas con una serie de edades: definíamos un vector *edad* con capacidad para almacenar la edad de MAX_PERSONAS y una variable *personas*, cuyo valor siempre era menor o igual que MAX_PERSONAS, nos indicaba cuántos elementos del vector contenían realmente datos. Hay algo poco elegante en esa solución: las variables *edad* y *personas* son variables independientes, que no están relacionadas entre sí en el programa (salvo por el hecho de que nosotros sabemos que sí lo están). Una solución más elegante pasa por crear un registro que contenga el número de personas y, en un vector, las edades. He aquí el programa que ya te presentamos en su momento convenientemente modificado según este nuevo principio de diseño:

edades_8.c — **edades.c**

```
#include <stdio.h>
#include <math.h>

#define MAX_PERSONAS 20

struct ListaEdades {
   int edad[MAX_PERSONAS];  // Vector con capacidad para MAX_PERSONAS edades.
   int talla;               // Número de edades realmente almacenadas.
};

int main(void)
{
   struct ListaEdades personas;
   int i, j, aux, suma_edad;
   float suma_desviacion, media, desviacion;
   int moda, frecuencia, frecuencia_moda, mediana;

   /* Lectura de edades */
   personas.talla = 0;
   do {
      printf("Introduce edad de %d (si es negativa, acabar): ",
             personas.talla+1);
      scanf("%d", &personas.edad[personas.talla]);
      personas.talla++;
   } while (personas.talla < MAX_PERSONAS && personas.edad[personas.talla-1] >= 0);
   personas.talla--;

   if (personas.talla > 0) {
      /* Cálculo de la media */
      suma_edad = 0;
      for (i=0; i<personas.talla; i++)
         suma_edad += personas.edad[i];
      media = suma_edad / (float) personas.talla;

      /* Cálculo de la desviacion típica */
      suma_desviacion = 0.0;
      for (i=0; i<personas.talla; i++)
         suma_desviacion += (personas.edad[i] - media) * (personas.edad[i] - media);
      desviacion = sqrt( suma_desviacion / personas.talla );

      /* Cálculo de la moda */
      for (i=0; i<personas.talla-1; i++)  // Ordenación mediante burbuja.
         for (j=0; j<personas.talla-i; j++)
            if (personas.edad[j] > personas.edad[j+1]) {
               aux = personas.edad[j];
               personas.edad[j] = personas.edad[j+1];
```

```
            personas.edad[j+1] = aux;
          }

    frecuencia = 0;
    frecuencia_moda = 0;
    moda = -1;
    for (i=0; i<personas.talla-1; i++)
      if (personas.edad[i] == personas.edad[i+1])
        if (++frecuencia > frecuencia_moda) {
          frecuencia_moda = frecuencia;
          moda = personas.edad[i];
        }
      else
        frecuencia = 0;

    /* Cálculo de la mediana */
    mediana = personas.edad[personas.talla/2];

    /* Impresión de resultados */
    printf("Edad media  : %f\n", media);
    printf("Desv. típica: %f\n", desviacion);
    printf("Moda        : %d\n", moda);
    printf("Mediana     : %d\n", mediana);
  }
  else
    printf("No se introdujo dato alguno.\n");

  return 0;
}
```

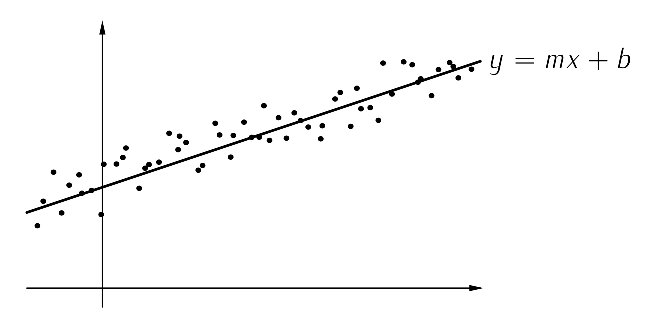
. . . . EJERCICIOS . . . .

▶ 143 Modifica el programa de cálculo con polinomios que sirvió de ejemplo en el apartado 2.1.5 para representar los polinomios mediante registros. Cada registro contendrá dos campos: el grado del polinomio y el vector con los coeficientes.

. . . .

2.4.2. Un ejemplo: rectas de regresión para una serie de puntos en el plano

Hay métodos estadísticos que permiten obtener una recta que se ajusta de forma óptima a una serie de puntos en el plano.



Si disponemos de una serie de n puntos (x_1, y_1), (x_2, y_2), . . . , (x_n, y_n), la recta de ajuste $y = mx + b$ que minimiza el cuadrado de la distancia vertical de todos los puntos a la

recta se puede obtener efectuando los siguientes cálculos:

$$m = \frac{\left(\sum_{i=1}^{n} x_i\right) \cdot \left(\sum_{i=1}^{n} y_i\right) - n \cdot \sum_{i=1}^{n} x_i y_i}{\left(\sum_{i=1}^{n} x_i\right)^2 - n \cdot \sum_{i=1}^{n} x_i^2},$$

$$b = \frac{\left(\sum_{i=1}^{n} y_i\right) \cdot \left(\sum_{i=1}^{n} x_i^2\right) - \left(\sum_{i=1}^{n} x_i\right) \cdot \left(\sum_{i=1}^{n} x_i y_i\right)}{n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}.$$

Las fórmulas asustan un poco, pero no contienen más que sumatorios. El programa que vamos a escribir lee una serie de puntos (con un número máximo de, pongamos, 1000), y muestra los valores de m y b.

Modelaremos los puntos con un registro:

```
struct Punto {
  float x, y;
};
```

El vector de puntos, al que en principio denominaremos p, tendrá talla 1000:

```
#define TALLAMAX 1000
struct Punto p[TALLAMAX];
```

Pero 1000 es el número máximo de puntos. El número de puntos disponibles efectivamente será menor o igual y su valor deberá estar accesible en alguna variable. Olvidémonos del vector p: nos conviene definir un registro en el que se almacenen vector y talla real del vector.

```
struct ListaPuntos {
  struct Punto punto[TALLAMAX];
  int talla;
};
```

Observa que estamos anidando **struct**s.

Necesitamos ahora una variable del tipo que hemos definido:

```
#include <stdio.h>

#define TALLAMAX 1000

struct Punto {
  float x, y;
};

struct ListaPuntos {
  struct Punto punto[TALLAMAX];
  int talla;
};

int main(void)
{
  struct ListaPuntos lista;
  ...
```

Reflexionemos brevemente sobre cómo podemos acceder a la información de la variable *lista*:

Expresión	Tipo y significado
lista	Es un valor de tipo **struct** *ListaPuntos*. Contiene un vector de 1000 puntos y un entero.
lista.*talla*	Es un entero. Indica cuántos elementos del vector contienen información.
lista.*punto*	Es un vector de 1000 valores de tipo **struct** *Punto*.
lista.*punto*[0]	Es el primer elemento del vector y es de tipo **struct** *Punto*, así que está compuesto por dos flotantes.
lista.*punto*[0].*x*	Es el campo *x* del primer elemento del vector. Su tipo es **float**.
lista.*punto*[*lista*.*talla*-1].*y*	Es el campo *y* del último elemento con información del vector. Su tipo es **float**.
lista.*punto*.*x*	¡Error! Si *lista*.*punto* es un vector, no podemos acceder al campo *x*.
lista.*punto*.*x*[0]	¡Error! Si lo anterior era incorrecto, ésto lo es aún más.
lista.*punto*.[0].*x*	¡Error! ¿Qué hace un punto antes del operador de indexación?
lista[0].*punto*	¡Error! La variable *lista* no es un vector, así que no puedes aplicar el operador de indexación sobre ella.

Ahora que tenemos más claro cómo hemos modelado la información, vamos a resolver el problema propuesto. Cada uno de los sumatorios se precalculará cuando se hayan leído los puntos. De ese modo, simplificaremos significativamente las expresiones de cálculo de m y b. Debes tener en cuenta que, aunque en las fórmulas se numeran los puntos empezando en 1, en C se empieza en 0.

Veamos el programa completo:

ajuste.c

```
#include <stdio.h>

#define TALLAMAX 1000

struct Punto {
  float x, y;
};

struct ListaPuntos {
  struct Punto punto[TALLAMAX];
  int talla;
};

int main(void)
{
  struct ListaPuntos lista;

  float sx, sy, sxy, sxx;
  float m, b;
  int i;

  /* Lectura de puntos */
  printf("Puntos␣a␣leer:␣"); scanf("%d", &lista.talla);
  for (i=0; i<lista.talla; i++) {
    printf("Coordenada␣x␣del␣punto␣%d:␣", i); scanf("%f", &lista.punto[i].x);
    printf("Coordenada␣y␣del␣punto␣%d:␣", i); scanf("%f", &lista.punto[i].y);
  }
```

```
  
  /* Cálculo de los sumatorios */
  sx = 0.0;
  for (i=0; i<lista.talla; i++)
    sx += lista.punto[i].x;

  sy = 0.0;
  for (i=0; i<lista.talla; i++)
    sy += lista.punto[i].y;

  sxy = 0.0;
  for (i=0; i<lista.talla; i++)
    sxy += lista.punto[i].x * lista.punto[i].y;

  sxx = 0.0;
  for (i=0; i<lista.talla; i++)
    sxx += lista.punto[i].x * lista.punto[i].x;

  /* Cálculo de m y b e impresión de resultados */
  if (sx * sx - lista.talla * sxx == 0)
    printf("Indefinida\n");
  else {
    m = (sx * sy - lista.talla * sxy) / (sx * sx - lista.talla * sxx);
    printf("m = %f\n", m);
    b = (sy * sxx - sx * sxy) / (lista.talla * sxx - sx * sx);
    printf("b = %f\n", b);
  }

  return 0;
}
```

EJERCICIOS

▶ 144 Diseña un programa que lea una lista de hasta 1000 puntos por teclado y los almacene en una variable (del tipo que tú mismo definas) llamada *representantes*. A continuación, irá leyendo nuevos puntos hasta que se introduzca el punto de coordenadas $(0, 0)$. Para cada nuevo punto, debes encontrar cuál es el punto más próximo de los almacenados en *representantes*. Calcula la distancia entre dos puntos como la distancia euclídea.

▶ 145 Deseamos efectuar cálculos con enteros positivos de hasta 1000 cifras, más de las que puede almacenar un **int** (o incluso **long long int**). Define un registro que permita representar números de hasta 1000 cifras con un vector en el que cada elemento es una cifra (representada con un **char**). Representa el número de cifras que tiene realmente el valor almacenado con un campo del registro. Escribe un programa que use dos variables del nuevo tipo para leer dos números y que calcule el valor de la suma y la resta de estos (supondremos que la resta siempre proporciona un entero positivo como resultado).

2.4.3. Otro ejemplo: gestión de una coleción de CDs

Estamos en condiciones de abordar la implementación de un programa moderadamente complejo: la gestión de una colección de CDs (aunque, todavía, sin poder leer/escribir en fichero). De cada CD almacenaremos los siguientes datos:

- el título (una cadena con, a lo sumo, 80 caracteres),
- el intérprete (una cadena con, a lo sumo, 40 caracteres),

- la duración (en minutos y segundos),
- el año de publicación.

Definiremos un registro para almacenar los datos de un CD y otro para representar la duración, ya que ésta cuenta con dos valores (minutos y segundos):

```
#define LONTITULO 80
#define LONINTERPRETE 40

struct Tiempo {
   int minutos;
   int segundos;
};

struct CompactDisc {
   char titulo[LONTITULO+1];
   char interprete[LONINTERPRETE+1];
   struct Tiempo duracion;
   int anyo;
};
```

Vamos a usar un vector para almacenar la colección, definiremos un máximo número de CDs: 1000. Eso no significa que la colección tenga 1000 discos, sino que puede tener a lo sumo 1000. ¿Y cuántos tiene en cada instante? Utilizaremos una variable para mantener el número de CDs presente en la colección. Mejor aún: definiremos un nuevo tipo de registro que represente a la colección entera de CDs. El nuevo tipo contendrá dos campos: el vector de discos (con capacidad limitada a 1000 unidades) y el número de discos en el vector. He aquí la definición de la estructura y la declaración de la colección de CDs:

```
#define MAXDISCOS 1000

...

struct Coleccion {
   struct CompactDisc cd[MAXDISCOS];
   int cantidad;
};

struct Coleccion mis_cds;
```

Nuestro programa permitirá efectuar las siguientes acciones:

- Añadir un CD a la base de datos.
- Listar toda la base de datos.
- Listar los CDs de un intérprete.
- Suprimir un CD dado su título y su intérprete.

(El programa no resultará muy útil hasta que aprendamos a utilizar ficheros en C, pues al finalizar cada ejecución se pierde toda la información registrada.)

He aquí el programa completo:

discoteca.c

```
#include <stdio.h>
#include <string.h>

#define MAXLINEA 80

```

```
#define MAXDISCOS 1000
#define LONTITULO 80
#define LONINTERPRETE 40

enum { Anyadir=1, ListadoCompleto, ListadoPorInterprete, Suprimir, Salir };

struct Tiempo {
  int minutos;
  int segundos;
};

struct CompactDisc {
  char titulo[LONTITULO+1];
  char interprete[LONINTERPRETE+1];
  struct Tiempo duracion;
  int anyo;
};

struct Coleccion {
  struct CompactDisc cd[MAXDISCOS];
  int cantidad;
};

int main(void)
{
  struct Coleccion mis_cds;
  int opcion, i, j;
  char titulo[LONTITULO+1], interprete[LONINTERPRETE+1];
  char linea[MAXLINEA]; // Para evitar los problemas de scanf.

  /* Inicialización de la colección. */
  mis_cds.cantidad = 0;

  /* Bucle principal: menú de opciones. */
  do {
    do {
      printf("Colección de CDs\n");
      printf("--------\n");
      printf("1) Añadir CD\n");
      printf("2) Listar todo\n");
      printf("3) Listar por intérprete\n");
      printf("4) Suprimir CD\n");
      printf("5) Salir\n");
      printf("Opción: ");
      gets(linea); sscanf(linea, "%d", &opcion);
      if (opcion <1 || opcion >5)
        printf("Opción inexistente. Debe estar entre 1 y 5\n");
    } while (opcion <1 || opcion >5);

    switch(opcion) {
      case Anyadir: // Añadir un CD.
        if (mis_cds.cantidad == MAXDISCOS)
          printf("La base de datos está llena. Lo siento.\n");
        else {
          printf("Título:     ");
          gets(mis_cds.cd[mis_cds.cantidad].titulo);
          printf("Intérprete: ");
          gets(mis_cds.cd[mis_cds.cantidad].interprete);
          printf("Minutos:    ");
```

```
            gets(linea);
            sscanf(linea, "%d", &mis_cds.cd[mis_cds.cantidad].duracion.minutos);
            printf("Segundos:␣␣␣");
            gets(linea);
            sscanf(linea, "%d", &mis_cds.cd[mis_cds.cantidad].duracion.segundos);
            printf("Año:␣␣␣␣␣␣␣␣");
            gets(linea);
            sscanf(linea, "%d", &mis_cds.cd[mis_cds.cantidad].anyo);
            mis_cds.cantidad++;
          }
          break;

        case ListadoCompleto: // Listar todo.
          for (i=0; i<mis_cds.cantidad; i++)
            printf("%d␣%s␣de␣%s␣(%d:%d)␣%d\n", i,
                mis_cds.cd[i].titulo,
              mis_cds.cd[i].interprete,
              mis_cds.cd[i].duracion.minutos,
              mis_cds.cd[i].duracion.segundos,
              mis_cds.cd[i].anyo);
          break;

        case ListadoPorInterprete: // Listar por intérprete.
          printf("Intérprete:␣"); gets(interprete);
          for (i=0; i<mis_cds.cantidad; i++)
            if (strcmp(interprete, mis_cds.cd[i].interprete) == 0)
              printf("%d␣%s␣de␣%s␣(%d:%d)␣%d\n", i,
                  mis_cds.cd[i].titulo,
                mis_cds.cd[i].interprete,
                mis_cds.cd[i].duracion.minutos,
                mis_cds.cd[i].duracion.segundos,
                mis_cds.cd[i].anyo);
          break;

        case Suprimir: // Suprimir CD.
          printf("Título:␣␣␣␣␣"); gets(titulo);
          printf("Intérprete:␣"); gets(interprete);
          for (i=0; i<mis_cds.cantidad; i++)
            if (strcmp(titulo, mis_cds.cd[i].titulo) == 0 &&
                strcmp(interprete, mis_cds.cd[i].interprete) == 0)
                break;
          if (i < mis_cds.cantidad) {
            for (j=i+1; j<mis_cds.cantidad; j++)
              mis_cds.cd[j-1] = mis_cds.cd[j];
            mis_cds.cantidad--;
          }
          break;
      }

  } while (opcion != Salir);
  printf("Gracias␣por␣usar␣nuestro␣programa.\n");

  return 0;
}
```

En nuestro programa hemos separado la definición del tipo **struct** *Coleccion* de la declaración de la variable *mis_cds*. No es necesario. Podemos definir el tipo y declarar la variable en una sola sentencia:

```
struct Coleccion {
```

```
    struct CompactDisc cd[MAXDISCOS];
    int cantidad;
} mis_cds; // Declara la variable mis_cds como de tipo struct Coleccion.
```

Apuntemos ahora cómo enriquecer nuestro programa de gestión de una colección de discos compactos almacenando, además, las canciones de cada disco. Empezaremos por definir un nuevo registro: el que modela una canción. De cada canción nos interesa el título, el autor y la duración:

```
struct Cancion {
    char titulo[LONTITULO+1];
    char autor[LONINTERPRETE+1];
    struct Tiempo duracion;
};
```

Hemos de modificar el registro **struct** *CompactDisc* para que almacene hasta, digamos, 20 canciones:

```
#define MAXCANCIONES 20

struct CompactDisc {
    char titulo[LONTITULO+1];
    char interprete[LONINTERPRETE+1];
    struct Tiempo duracion;
    int anyo;
    struct Cancion cancion[MAXCANCIONES]; // Vector de canciones.
    int canciones; // Número de canciones que realmente hay.
};
```

¿Cómo leemos ahora un disco compacto? Aquí tienes, convenientemente modificada, la porción del programa que se encarga de ello:

```
...
int main(void)
{
    int segundos;
    ...
        switch(opcion) {
            case Anyadir: // Añadir un CD.
                if (mis_cds.cantidad == MAXDISCOS)
                    printf("La base de datos está llena. Lo siento.\n");
                else {
                    printf("Título:     ");
                    gets(mis_cds.cd[mis_cds.cantidad].titulo);
                    printf("Intérprete: ");
                    gets(mis_cds.cd[mis_cds.cantidad].interprete);
                    printf("Año:        ");
                    gets(linea); sscanf(linea, "%d", &mis_cds.cd[mis_cds.cantidad].anyo);

                    do {
                        printf("Número de canciones: ");
                        gets(linea);
                        sscanf(linea, "%d", &mis_cds.cd[mis_cds.cantidad].canciones);
                    } while (mis_cds.cd[mis_cds.cantidad].canciones > MAXCANCIONES);

                    for (i=0; i<mis_cds.cd[mis_cds.cantidad].canciones; i++) {
                        printf("Título de la canción número %d: ", i);
                        gets(mis_cds.cd[mis_cds.cantidad].cancion[i].titulo);
                        printf("Autor de la canción número %d: ", i);
                        gets(mis_cds.cd[mis_cds.cantidad].cancion[i].autor);
                        printf("Minutos que dura la canción número %d: ", i);
```

```
                gets(linea);
                sscanf(linea, "%d",
                        &mis_cds.cd[mis_cds.cantidad].cancion[i].duracion.minutos);
                printf("y␣segundos:␣");
                gets(linea);
                sscanf(linea, "%d",
                        &mis_cds.cd[mis_cds.cantidad].cancion[i].duracion.segundos);
              }

              segundos = 0;
              for (i=0; i<mis_cds.cd[mis_cds.cantidad].canciones; i++)
                segundos +=60 * mis_cds.cd[mis_cds.cantidad].cancion[i].duracion.minutos
                           + mis_cds.cd[mis_cds.cantidad].cancion[i].duracion.segundos;
              mis_cds.cd[mis_cds.cantidad].duracion.minutos = segundos / 60;
              mis_cds.cd[mis_cds.cantidad].duracion.segundos = segundos % 60;

              mis_cds.cantidad++;
            }
            break;
  ...
}
```

Observa cómo se calcula ahora la duración del compacto como suma de las duraciones de todas sus canciones.

. . . . EJERCICIOS . . . .

▶ **146** Diseña un programa C que gestione una agenda telefónica. Cada entrada de la agenda contiene el nombre de una persona y *hasta* 10 números de teléfono. El programa permitirá añadir nuevas entradas a la agenda y nuevos teléfonos a una entrada ya existente. El menú del programa permitirá, además, borrar entradas de la agenda, borrar números de teléfono concretos de una entrada y efectuar búsquedas por las primeras letras del nombre. (Si, por ejemplo, tu agenda contiene entradas para «José Martínez», «Josefa Pérez» y «Jaime Primero», una búsqueda por «Jos» mostrará a las dos primeras personas y una búsqueda por «J» las mostrará a todas.)

. . . .

2.5. Definición de nuevos tipos de datos

Los registros son nuevos tipos de datos cuyo nombre viene precedido por la palabra **struct**. C permite definir nuevos nombres para los tipos existentes con la palabra clave **typedef**.

He aquí un posible uso de **typedef**:

```
#define LONTITULO 80
#define LONINTERPRETE 40

struct Tiempo {
  int minutos;
  int segundos;
};

typedef struct Tiempo TipoTiempo;

struct Cancion {
  char titulo[LONTITULO+1];
  char autor[LONINTERPRETE+1];
  TipoTiempo duracion;
};
```

```
typedef struct Cancion TipoCancion;


struct CompactDisc {
  char titulo[LONTITULO+1];
  char interprete[LONINTERPRETE+1];
  TipoTiempo duracion;
  int anyo;
  TipoCancion cancion[MAXCANCIONES]; // Vector de canciones.
  int canciones; // Número de canciones que realmente hay.
};
```

Hay una forma más compacta de definir un nuevo tipo a partir de un registro:

```
#define LONTITULO 80
#define LONINTERPRETE 40

typedef struct {
  int minutos;
  int segundos;
} TipoTiempo;

typedef struct {
  char titulo[LONTITULO+1];
  char autor[LONINTERPRETE+1];
  TipoTiempo duracion;
} TipoCancion;

typedef struct {
  char titulo[LONTITULO+1];
  char interprete[LONINTERPRETE+1];
  TipoTiempo duracion;
  int anyo;
  TipoCancion cancion[MAXCANCIONES]; // Vector de canciones.
  int canciones; // Número de canciones que realmente hay.
} TipoCompactDisc;

typedef struct {
  TipoCompactDisc cd[MAXDISCOS];
  int    cds;
} TipoColeccion;

int main(void)
{
  TipoColeccion mis_cds;
  ...
```

Observa que, sistemáticamente, hemos utilizado iniciales mayúsculas para los nombres de tipos de datos (definidos con **typedef** y **struct** o sólo con **struct**). Es un buen convenio para no confundir variables con tipos. Te recomendamos que hagas lo mismo o, en su defecto, que adoptes cualquier otro criterio, pero que sea coherente. El renombramiento de tipos no sólo sirve para eliminar la molesta palabra clave **struct**, también permite diseñar programas más legibles y en los que resulta más fácil cambiar tipos globalmente.

Imagina que en un programa nuestro representamos la edad de una persona con un valor entre 0 y 127 (un **char**). Una variable *edad* se declararía así:

```
char edad;
```

No es muy elegante: una edad no es un carácter, sino un número. Si definimos un «nuevo» tipo, el programa es más legible:

```
typedef char TipoEdad;

TipoEdad edad;
```

Es más, si más adelante deseamos cambiar el tipo **char** por **int**, sólo hemos de cambiar la línea que empieza por **typedef**, aunque hayamos definido decenas de variables del tipo **TipoEdad**:

```
typedef int TipoEdad;

TipoEdad edad;
```

> ***Los cambios de tipos y sus consecuencias***
>
> Hemos dicho que **typedef** define nuevos tipos y facilita sustituir un tipo por otro en diferentes versiones de un programa. Es cierto, pero problemático. Considera que definimos un tipo edad como carácter sin signo y que definimos una variable de dicho tipo cuyo valor leemos de teclado:
>
> ```
> #include <stdio.h>
>
> typedef unsigned char TipoEdad;
>
> int main(void)
> {
>   TipoEdad mi_edad;
>
>   scanf("%hhu", &mi_edad);
>   printf("Valor␣leído␣%hhu\n", mi_edad);
>   return 0;
> }
> ```
>
> ¿Qué pasa si, posteriormente, decidimos que el **TipoEdad** debiera ser un entero de 32 bits? He aquí una versión errónea del programa:
>
> ```
> #include <stdio.h>
>
> typedef int TipoEdad;
>
> int main(void)
> {
>   TipoEdad mi_edad;
>
>   printf("Introduzca␣edad:␣");
>   scanf("%hhu", &mi_edad); // ¡Mal!
>   printf("Valor␣leído␣%hhu\n", mi_edad); // ¡Mal!
>   return 0;
> }
> ```
>
> ¿Y por qué es erróneo? Porque debiéramos haber modificado *además* las marcas de formato de *scanf* y *printf*: en lugar de `%hhu` deberíamos usar ahora `%d`. C no es un lenguaje idóneo para este tipo de modificaciones. Otros lenguajes, como C++ soportan de forma mucho más flexible la posibilidad de cambiar tipos de datos, ya que no obligan al programador a modificar un gran número de líneas del programa.

Capítulo 3

Funciones

Un momento después, Alicia atravesaba el cristal, y saltaba ágilmente a la habitación del Espejo.

LEWIS CARROLL, *Alicia a través del espejo.*

Vamos a estudiar la definición y uso de funciones en C. El concepto es el mismo que ya estudiaste al aprender Python: una función es un fragmento de programa parametrizado que efectúa unos cálculos y, o devuelve un valor como resultado, o tiene efectos laterales (modificación de variables globales o argumentos, volcado de información en pantalla, etc.), o ambas cosas. La principal diferencia entre Python y C estriba en el paso de parámetros. En este aspecto, C presenta ciertas limitaciones frente a Python, pero también ciertas ventajas. Entre las limitaciones tenemos la necesidad de dar un tipo a cada parámetro y al valor de retorno, y entre las ventajas, la posibilidad de pasar variables escalares y modificar su valor en el cuerpo de la función (gracias al uso de punteros).

Estudiaremos también la posibilidad de declarar y usar variables locales, y volveremos a tratar la recursividad. Además, veremos cómo implementar nuestros propios módulos mediante las denominadas unidades de compilación y la creación de ficheros de cabecera.

Finalmente, estudiaremos la definición y el uso de macros, una especie de «pseudo-funciones» que gestiona el preprocesador de C.

3.1. Definición de funciones

En C no hay una palabra reservada (como **def** en Python) para iniciar la definición de una función. El aspecto de una definición de función en C es éste:

```
tipo_de_retorno identificador ( parámetros )
{
  cuerpo_de_la_función
}
```

El cuerpo de la función puede contener declaraciones de variables locales (típicamente en sus primeras líneas).

Aquí tienes un ejemplo de definición de función: una función que calcula el logaritmo en base *b* (para *b* entero) de un número *x*. La hemos definido de un modo menos compacto de lo que podemos hacer para ilustrar los diferentes elementos que puedes encontrar en una función:

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

```

```
    logbase = log10(b);
    resultado = log10(x)/logbase;
    return resultado;
}
```

Detengámonos a analizar brevemente cada uno de los componentes de la definición de una función e identifiquémoslos en el ejemplo:

- El *tipo de retorno* indica de qué tipo de datos es el valor devuelto por la función como resultado (más adelante veremos cómo definir procedimientos, es decir, funciones sin valor de retorno). Puedes considerar esto como una limitación frente a Python: en C, cada función devuelve valores de un único tipo. No podemos definir una función que, según convenga, devuelva un entero, un flotante o una cadena, como hicimos en Python cuando nos convino.

  En nuestro ejemplo, la función devuelve un valor de tipo **float**.

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}
```

- El *identificador* es el nombre de la función y, para estar bien formado, debe observar las mismas reglas que se siguen para construir nombres de variables. Eso sí, no puedes definir una función con un identificador que ya hayas usado para una variable (u otra función).

  El identificador de nuestra función de ejemplo es *logaritmo*:

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}
```

- Entre paréntesis aparece una lista de declaraciones de *parámetros* separadas por comas. Cada declaración de parámetro indica tanto el tipo del mismo como su identificador[1].

  Nuestra función tiene dos parámetros, uno de tipo **float** y otro de tipo **int**.

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}
```

[1] Eso en el caso de parámetros escalares. Los parámetros de tipo vectorial se estudiarán más adelante.

- El *cuerpo* de la función debe ir encerrado entre llaves, aunque sólo conste de una sentencia. Puede empezar por una declaración de variables locales a la que sigue una o más sentencias C. La sentencia **return** permite finalizar la ejecución de la función y devolver un valor (que debe ser del mismo tipo que el indicado como tipo de retorno). Si no hay sentencia **return**, la ejecución de la función finaliza también al acabar de ejecutar la última de las sentencias de su cuerpo, pero es un error no devolver nada con **return** si se ha declarado la función como tal, y no como procedimiento.

  Nuestra función de ejemplo tiene un cuerpo muy sencillo. Hay una declaración de variables (locales) y está formado por tres sentencias, dos de asignación y una de devolución de valor:

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}
```

- La *sentencia (o sentencias) de devolución de valor* forma(n) parte del cuerpo y empieza(n) con la palabra **return**. Una función puede incluir más de una sentencia de devolución de valor, pero debes tener en cuenta que la ejecución de la función finaliza con la primera ejecución de una sentencia **return**.

```
float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}
```

La función *logaritmo* se invoca como una función cualquiera de `math.h`:

logaritmo.c

logaritmo.c

```
#include <stdio.h>
#include <math.h>

float logaritmo (float x, int b)
{
  float logbase, resultado;

  logbase = log10(b);
  resultado = log10(x)/logbase;
  return resultado;
}

int main (void)
{
  float y;

  y = logaritmo(128.0, 2);
  printf("%f\n", y);

```

```
    return 0;
}
```

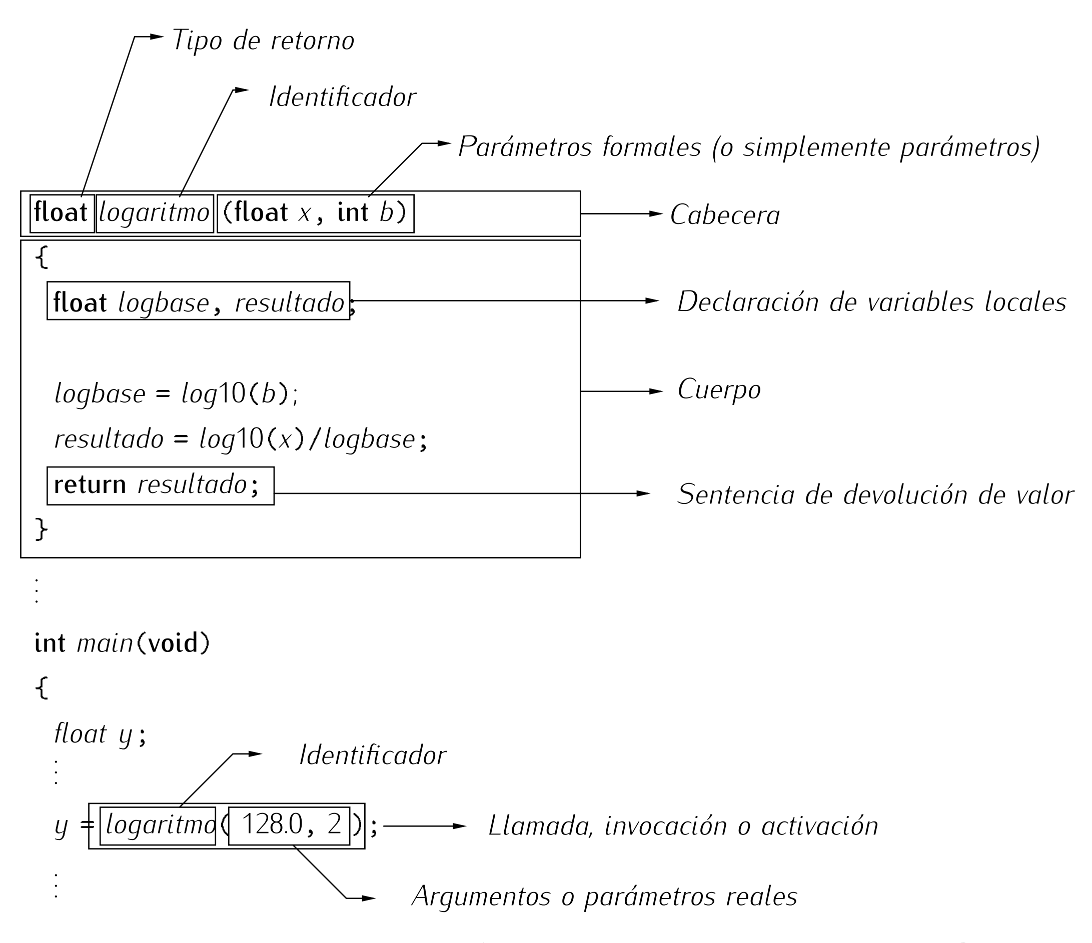
Si ejecutamos el programa tenemos:

```
7.000000
```

Es necesario que toda función se defina en el programa antes de la primera línea en que se usa. Por esta razón, todas nuestras funciones se definen delante de la función *main*, que es la función que contiene el programa principal y a la que, por tanto, no se llama desde ningún punto del programa.[2]

Naturalmente, ha resultado necesario incluir la cabecera `math.h` en el programa, ya que usamos la función *log*10. Recuerda, además, que al compilar se debe enlazar con la biblioteca matemática, es decir, se debe usar la opción `-lm` de `gcc`.

Esta ilustración te servirá para identificar los diferentes elementos de la definición de una función y de su invocación:



¡Ah! Te hemos dicho antes que la función *logaritmo* no es muy compacta. Podríamos haberla definido así:

```
float logaritmo (float x, int b)
{
  return log10(x)/log10(b);
}
```

EJERCICIOS

▶ 147 Define una función que reciba un **int** y devuelva su cuadrado.

[2]Nuevamente hemos de matizar una afirmación: en realidad sólo es necesario que se haya declarado el *prototipo* de la función. Más adelante daremos más detalles.

▶ **148** Define una función que reciba un **float** y devuelva su cuadrado.

▶ **149** Define una función que reciba dos **float** y devuelva 1 («cierto») si el primero es menor que el segundo y 0 («falso») en caso contrario.

▶ **150** Define una función que calcule el volumen de una esfera a partir de su radio r. (Recuerda que el volumen de una esfera de radio r es $4/3\pi r^3$.)

▶ **151** El seno hiperbólico de x es

$$\sinh = \frac{e^x - e^{-x}}{2}.$$

Diseña una función C que efectúe el calculo de senos hiperbólicos. (Recuerda que e^x se puede calcular con la función *exp*, disponible incluyendo `math.h` y enlazando el programa ejecutable con la librería matemática.)

▶ **152** Diseña una función que devuelva «cierto» (el valor 1) si el año que se le suministra como argumento es bisiesto, y «falso» (el valor 0) en caso contrario.

▶ **153** La distancia de un punto (x_0, y_0) a una recta $Ax + By + C = 0$ viene dada por

$$d = \frac{Ax_0 + By_0 + C}{\sqrt{A^2 + B^2}}.$$

Diseña una función que reciba los valores que definen una recta y los valores que definen un punto y devuelva la distancia del punto a la recta.

. . . .

Veamos otro ejemplo de definición de función:

```c
int minimo(int a, int b, int c)
{
   if (a <= b)
      if (a <= c)
         return a;
      else
         return c;
   else
      if (b <= c)
         return b;
      else
         return c;
}
```

La función *minimo* devuelve un dato de tipo **int** y recibe tres datos, también de tipo **int**. No hay problema en que aparezca más de una sentencia **return** en una función. El comportamiento de **return** es el mismo que estudiamos en Python: tan pronto se ejecuta, finaliza la ejecución de la función y se devuelve el valor indicado.

Observa que *main* es una función. Su cabecera es **int** *main*(**void**). ¿Qué significa **void**? Significa que no hay parámetros. Pero no nos adelantemos. En este mismo capítulo hablaremos de funciones sin parámetros.

. . . . EJERCICIOS . . . .

▶ **154** Define una función que, dada una letra minúscula del alfabeto inglés, devuelva su correspondiente letra mayúscula. Si el carácter recibido como dato no es una letra minúscula, la función la devolverá inalterada.

▶ **155** ¿Qué error encuentras en esta función?

```
int minimo (int a, b, c)
{
  if (a <= b && a <= c)
    return a;
  if (b <= a && b <= c)
    return b;
  return c;
}
```

3.2. Variables locales y globales

Cada función puede definir sus propias variables locales definiéndolas en su cuerpo. C permite, además, definir variables fuera del cuerpo de cualquier función: son las variables globales.

3.2.1. Variables locales

Las variables que declaramos justo al principio del cuerpo de una función son *variables locales*. Este programa, por ejemplo, declara dos variables locales para calcular el sumatorio $\sum_{i=a}^{b} i$. La variable local a *sumatorio* con identificador *i* nada tiene que ver con la variable del mismo nombre que es local a *main*:

locales.c

```
#include <stdio.h>

int sumatorio(int a, int b)
{
  int i, s; // Variables locales a sumatorio.

  s = 0;
  for (i=a; i<=b; i++)
    s += i;
  return s;
}

int main(void)
{
  int i; // Variable local a main.

  for (i=1; i<=10; i++)
    printf("Suma de %d primeros números naturales: %d\n", i, sumatorio(1, i));
  return 0;
}
```

Las variables locales *i* y *s* de *sumatorio* sólo «viven» durante las llamadas a *sumatorio*.

La zona en la que es visible una variable es su *ámbito*. Las variables locales sólo son visibles en el cuerpo de la función en la que se declaran; ése es su ámbito.

EJERCICIOS

▶ 156 Diseña una función que calcule el factorial de un entero n.

▶ 157 Diseña una función que calcule x^n, para n entero y x de tipo **float**. (Recuerda que si n es negativo, x^n es el resultado de multiplicar $1/x$ por sí mismo $-n$ veces.)

Variables locales a bloques

El concepto de variable local no está limitado, en C, a las funciones. En realidad, puedes definir variables locales en cualquier bloque de un programa. Fíjate en este ejemplo:

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i=0; i<3; i++) {
    int j;
    for (j=0; j<3; j++)
      printf("%d-%d␣␣", i, j);
    printf("\n");
  }
  return 0;
}
```

La variable *j* sólo existe en el bloque en el que se ha declarado, es decir, en la zona sombreada. Ese es su ámbito. La variable *i* tiene un ámbito que engloba al de *j*.

Puedes comprobar, pues, que una variable local a una función es también una variable local a un bloque: sólo existe en el bloque que corresponde al cuerpo de la función.

Como ya te dijimos en un cuadro del capítulo 1, C99 permite declarar variables de índice de bucle de usar y tirar. Su ámbito se limita al bucle. Aquí tienes un ejemplo en el que hemos sombreado el ámbito de la variable *j*:

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i=0; i<3; i++) {
    for (int j=0; j<3; j++)
      printf("%d-%d␣␣", i, j);
    printf("\n");
  }
  return 0;
}
```

▶ **158** El valor de la función e^x puede aproximarse con el desarrollo de Taylor:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Diseña una función que aproxime el valor de e^x usando n términos del desarrollo de Taylor, siendo n un número entero positivo. (Puedes usar, si te conviene, la función de exponenciación del último ejercicio para calcular los distintos valores de x^i, aunque hay formas más eficientes de calcular $x/1!$, $x^2/2!$, $x^3/3!$, …, ¿sabes cómo? Plantéate cómo generar un término de la forma $x^i/i!$ a partir de un término de la forma $x^{i-1}/(i-1)!$.)

▶ **159** El valor de la función coseno puede aproximarse con el desarrollo de Taylor:

$$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Andrés Marzal/Isabel Gracia - ISBN: 978-84-693-0143-2

Diseña una función que aproxime el coseno de un valor x usando n términos del desarrollo de Taylor, siendo n un número entero positivo.

▶ 160 Diseña una función que diga si un número es perfecto o no. Si el número es perfecto, devolverá «cierto» (el valor 1) y si no, devolverá «falso» (el valor 0). Un número es perfecto si es igual a la suma de todos sus divisores (excepto él mismo).

▶ 161 Diseña una función que diga si un número entero es o no es capicúa.

3.2.2. Variables globales

Las variables globales se declaran fuera del cuerpo de cualquier función y son accesibles desde cualquier punto del programa posterior a su declaración. Este fragmento de programa, por ejemplo, define una variable global *i* y una variable local a *main* con el mismo identificador:

globales.c   **globales.c**

```c
#include <stdio.h>

int i = 1;   // Variable global i.

int doble(void)
{
  i *= 2;   // Referencia a la variable global i.
  return i; // Referencia a la variable global i.
}

int main(void)
{
  int i; // Variable local i.

  for (i=0; i<5; i++) // Referencias a la variable local i.
    printf("%d\n", doble()); // Ojo: el valor mostrado corresponde a la i global.

  return 0;
}
```

Fíjate en la pérdida de legibilidad que supone el uso del identificador *i* en diferentes puntos del programa: hemos de preguntarnos siempre si corresponde a la variable local o global. Te desaconsejamos el uso generalizado de variables globales en tus programas. Como evitan usar parámetros en funciones, llegan a resultar muy cómodas y es fácil que abuses de ellas. No es que siempre se usen mal, pero se requiere una cierta experiencia para formarse un criterio firme que permita decidir cuándo resulta conveniente usar una variable global y cuándo conviene suministrar información a funciones mediante parámetros.

Como estudiante te pueden parecer un recurso cómodo para evitar suministrar información a las funciones mediante parámetros. Ese pequeño beneficio inmediato es, créenos, un lastre a medio y largo plazo: aumentará la probabilidad de que cometas errores al intentar acceder o modificar una variable y las funciones que definas en un programa serán difícilmente reutilizables en otros. Estás aprendiendo a programar y pretendemos evitar que adquieras ciertos vicios, así que te prohibimos que las uses... salvo cuando convenga que lo hagas.

EJERCICIOS

▶ 162 ¿Qué muestra por pantalla este programa?

contador_global.c

```
#include <stdio.h>

int contador; // Variable global.

void fija(int a)
{
  contador = a;
}

int decrementa(int a)
{
  contador -= a;
  return contador;
}

void muestra(int contador)
{
  printf("[%d]\n", contador);
}

void cuenta_atras(int a)
{
  int contador;
  for (contador=a; contador >=0; contador--)
    printf("%d ", contador);
  printf("\n");
}

int main(void) {
  int i;

  contador = 10;
  i = 1;
  while (contador >= 0) {
    muestra(contador);
    cuenta_atras(contador);
    muestra(i);
    decrementa(i);
    i *= 2;
  }
}
```

3.3. Funciones sin parámetros

Puedes definir una función sin parámetros dejando la palabra **void** como contenido de la lista de parámetros. Esta función definida por nosotros, por ejemplo, utiliza la función *rand* de `stdlib.h` para devolver un número aleatorio entre 1 y 6:

```
int dado (void)
{
  return rand() % 6 + 1;
}
```

Para llamar a la función *dado* hemos de añadir un par de paréntesis a la derecha del identificador, aunque no tenga parámetros.

Ya te habíamos anticipado que la función *main* es una función sin parámetros que devuelve un entero:

dado.c

```
#include <stdio.h>
#include <stdlib.h>

int dado(void)
{
  return rand() % 6 + 1;
}

int main(void)
{
  int i;
  for (i=0; i<10; i++)
    printf("%d\n", dado() );
  return 0;
}
```

«Calidad» de los números aleatorios

La función *rand* está pobremente implementada en muchas máquinas y genera patrones repetitivos que hacen poco aleatoria la secuencia de números generada. Este problema se agudiza si observamos los bits menos significativos de los números generados... ¡y eso es, precisamente, lo que estamos haciendo con expresiones como *rand*() % 6! Una forma de paliar este problema es usar una expresión diferente:

```
int dado(void)
{
  return (int) ((double) rand() / ((double) RAND_MAX + 1) * 6) + 1;
}
```

La constante `RAND_MAX` es el mayor número aleatorio que puede devolver *rand*. La división hace que el número generado esté en el intervalo $[0, 1[$.

EJERCICIOS

▶ 163 El programa `dado.c` siempre genera la misma secuencia de números aleatorios. Para evitarlo, debes proporcionar una semilla diferente con cada ejecución del programa. El valor de la semilla se suministra como único argumento de la función *srand*. Modifica `dado.c` para que solicite al usuario la introducción del valor semilla.

Un uso típico de las funciones sin parámetros es la lectura de datos por teclado que deben satisfacer una serie de restricciones. Esta función, por ejemplo, lee un número entero de teclado y se asegura de que sea par:

```
int lee_entero_par(void)
{
  int numero;

  scanf("%d", &numero);
  while (numero % 2 != 0) {
    printf("El número debe ser par y %d no lo es.\n", numero);
    numero = scanf("%d", &numero);
  }
  return numero;
}
```

Otro uso típico es la presentación de menús de usuario con lectura de la opción seleccionada por el usuario:

```
int menu_principal(void)
{
  int opcion;

  do {
    printf("1)␣Alta␣usuario\n");
    printf("2)␣Baja␣usuario\n");
    printf("3)␣Consulta␣usuario\n");
    printf("4)␣Salir\n");

    printf("Opción:␣"); scanf("%d", &opcion);
    if (opcion < 1 || opcion > 4)
      printf("Opción␣no␣válida.\n");
  } while (opcion < 1 || opcion > 4);

  return opcion;
}
```

EJERCICIOS

▶ 164 Diseña una función que lea por teclado un entero positivo y devuelva el valor leído. Si el usuario introduce un número negativo, la función advertirá del error por pantalla y leerá nuevamente el número cuantas veces sea menester.

3.4. Procedimientos

Un procedimiento, como recordarás, es una función que no devuelve valor alguno. Los procedimientos provocan efectos laterales, como imprimir un mensaje por pantalla, modificar variables globales o modificar el valor de sus parámetros.

Los procedimientos C se declaran como funciones con tipo de retorno **void**. Mira este ejemplo:

```
#include <stdio.h>

void saludos(void)
{
  printf("Hola,␣mundo.\n");
}
```

En un procedimiento puedes utilizar la sentencia **return**, pero sin devolver valor alguno. Cuando se ejecuta una sentencia **return**, finaliza inmediatamente la ejecución del procedimiento.

EJERCICIOS

▶ 165 Diseña un procedimiento que reciba un entero *n* y muestre por pantalla *n* asteriscos seguidos con un salto de línea al final.

▶ 166 Diseña un procedimiento que, dado un valor de *n*, dibuje con asteriscos un triángulo rectángulo cuyos catetos midan *n* caracteres. Si *n* es 5, por ejemplo, el procedimiento mostrará por pantalla este texto:

```
*
**
***
****
*****
```

Puedes usar, si te conviene, el procedimiento desarrollado en el ejercicio anterior.

▶ 167 Diseña un procedimiento que reciba un número entero entre 0 y 99 y muestre por pantalla su trascripción escrita. Si le suministramos, por ejemplo, el valor 31, mostrará el texto «`treinta y uno`».

3.5. Paso de parámetros

3.5.1. Parámetros escalares: paso por valor

Aunque modifiques el valor de un parámetro *escalar* en una función o procedimiento, el valor de la variable pasada como argumento permanece inalterado. La función *bits* del siguiente programa, por ejemplo, modifica en su cuerpo el valor de su parámetro *num*:

numbits.c

```c
#include <stdio.h>

int bits(unsigned int num)
{
  int b = 0;

  do {
    b++;
    num /= 2;
  } while (num > 0);

  return b;
}

int main(void)
{
  unsigned int numero;
  int bitsnumero;

  printf("Introduce un entero positivo: "); scanf("%u", &numero);
  bitsnumero = bits(numero);
  printf("Hay %d bits en %u.\n", bitsnumero, numero);
  return 0;
}
```

Al ejecutar el programa y teclear el número 128 se muestra por pantalla lo siguiente:

```
Introduce un entero positivo: 128
Hay 8 bits en 128.
```

Como puedes ver, el valor de *numero* permanece inalterado tras la llamada a *bits*, aunque en el cuerpo de la función se modifica el valor del parámetro *num* (que toma el valor de *numero* en la llamada). Un parámetro es como una variable local, sólo que su valor inicial se obtiene copiando el valor del argumento que suministramos. Así pues, *num* no es *numero*, sino otra variable que contiene *una copia* del valor de *numero*. Es lo que se denomina *paso de parámetro por valor*.

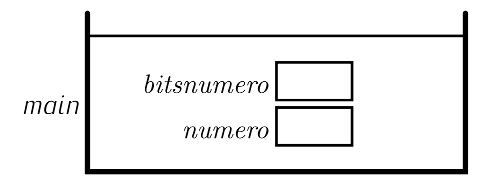
Llegados a este punto conviene que nos detengamos a estudiar cómo se gestiona la memoria en las llamadas a función.

3.5.2. Organización de la memoria: la pila de llamadas a función

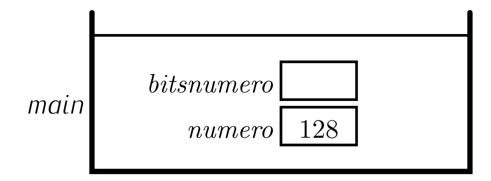
En C las variables locales se gestionan, al igual que en Python, mediante una pila. Cada función activada se representa en la pila mediante un *registro de activación* o *trama de*

activación. Se trata de una zona de memoria en la que se almacenan las variables locales y parámetros junto a otra información, como el punto desde el que se llamó a la función.

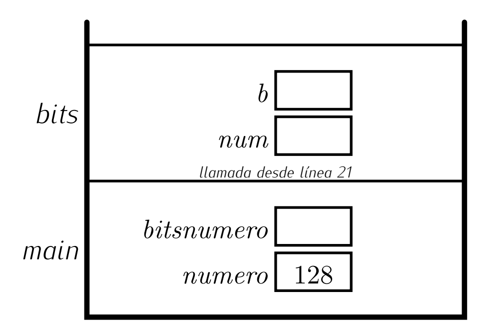
Cuando iniciamos la ejecución del programa `numbits.c`, se activa automáticamente la función *main*. En élla tenemos dos variables locales: *numero* y *bitsnumero*.



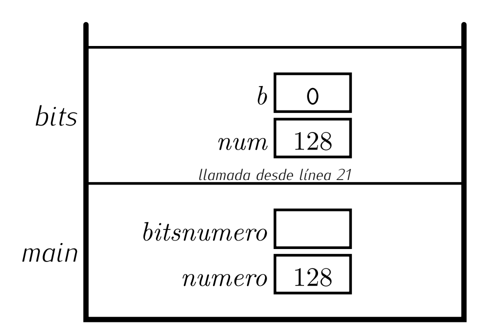
Si el usuario teclea el valor 128, éste se almacena en *numero*:



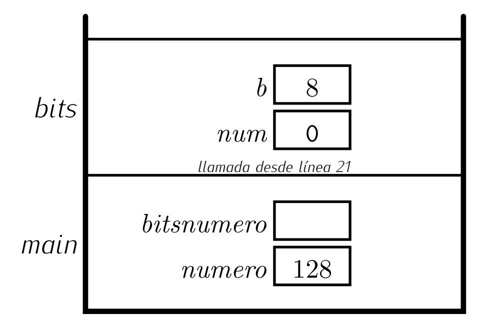
Cuando se produce la llamada a la función *bits*, se crea una nueva trama de activación:



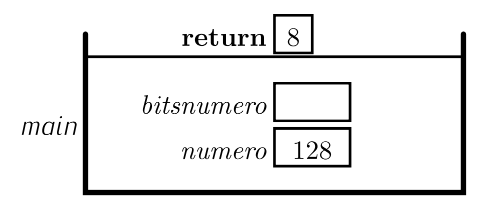
El parámetro *num* recibe una copia del contenido de *numero* y se inicializa la variable local *b* con el valor 0:



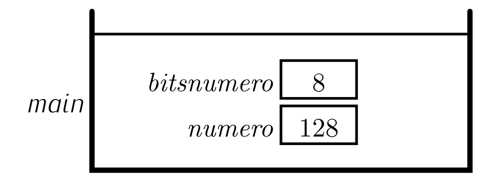
Tras ejecutar el bucle de *bits*, la variable *b* vale 8. Observa que aunque *num* ha modificado su valor y éste provenía originalmente de *numero*, el valor de *numero* no se altera:



La trama de activación de *bits* desaparece ahora, pero dejando constancia del valor devuelto por la función:



Y, finalmente, el valor devuelto se copia en *bitsnumero*:



Como ves, las variables locales sólo «viven» durante la ejecución de cada función. C obtiene *una copia* del valor de cada parámetro y la deja en la pila. Cuando modificamos el valor de un parámetro en el cuerpo de la función, estamos modificando el valor de la copia, no el de la variable original.

Este otro programa declara *numero* como una variable global y trabaja directamente con dicha variable:

numbits2.c

```
#include <stdio.h>

unsigned int numero;

int bits(void)
{
  int b = 0;

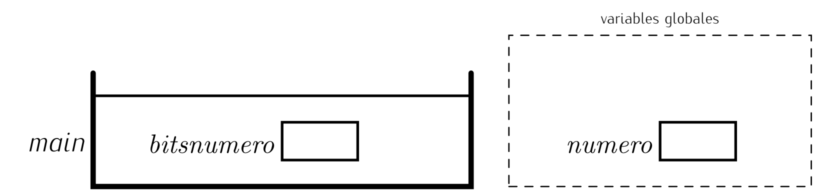
  do {
    b++;
    numero /= 2;
  } while (numero > 0);

  return b;
}

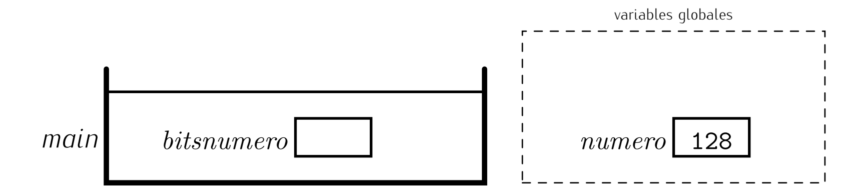
int main(void)
{
  int bitsnumero;

  printf("Introduce un entero positivo: "); scanf("%u", &numero);
  bitsnumero = bits();
  printf("Hay %d bits, pero ahora 'numero' vale %u.\n", bitsnumero, numero);
  return 0;
}
```

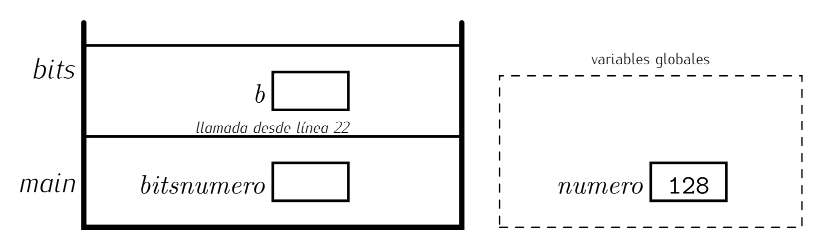
Las variables globales residen en una zona especial de la memoria y son accesibles desde cualquier función. Representaremos dicha zona como un área enmarcada con una línea discontínua. Cuando se inicia la ejecución del programa, ésta es la situación:



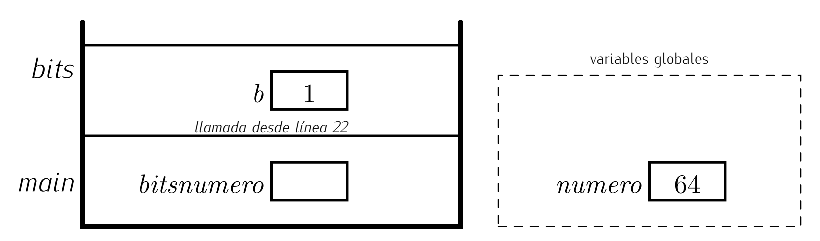
En *main* se da valor a la variable global *numero*:



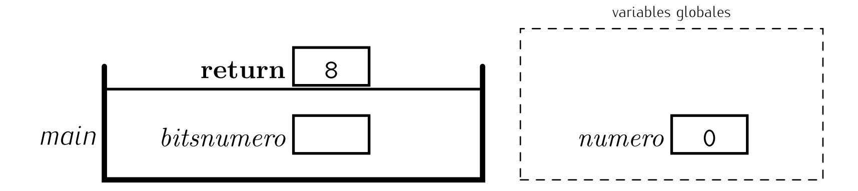
Y se llama a continuación a *bits* sin argumento alguno:



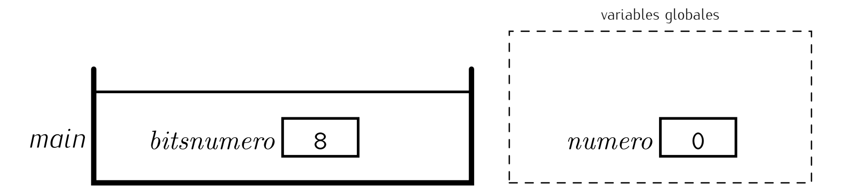
El cálculo de *bits* modifica el valor de *numero*. Tras la primera iteración del bucle **while**, ésta es la situación:



Cuando finaliza la ejecución de *bits* tenemos:



Entonces se copia el valor devuelto en *bitsnumero*:



El mensaje que obtenemos en pantalla es:

```
Introduce un entero positivo: 128
Hay 8 bits, pero ahora 'numero' vale 0.
```

Bueno. Ahora sabes qué pasa con las variables globales y cómo acceder a ellas desde las funciones. Pero repetimos lo que te dijimos al aprender Python: pocas veces está justificado acceder a variables globales, especialmente cuando estás aprendiendo. Evítalas.

. . . . EJERCICIOS . . . .

▶ **168** Estudia este programa y muestra gráficamente el contenido de la memoria cuando se van a ejecutar por primera vez las líneas 24, 14 y 5.

suma_cuadrados.c suma_cuadrados.c

```
#include <stdio.h>

int cuadrado(int i)
{
  return i * i;
}

int sumatorio(int a, int b)
{
  int i, s;

  s = 0;
  for (i=a; i<=b; i++)
    s += cuadrado(i);
  return s;
}

int main(void)
{
  int i, j;

  i = 10;
  j = 20;
  printf("%d\n", sumatorio(i, j));
  return 0;
}
```

▶ 169 Este programa muestra por pantalla los 10 primeros números primos. La función *siguiente* genera cada vez un número primo distinto. Gracias a la variable global *ultimoprimo* la función «recuerda» cuál fue el último número primo generado. Haz una traza paso a paso del programa (hasta que haya generado los 4 primeros primos). Muestra el estado de la pila y el de la zona de variables globales en los instantes en que se llama a la función *siguienteprimo* y cuando ésta devuelve su resultado

diez_primos.c diez_primos.c

```
#include <stdio.h>

int ultimoprimo = 0;

int siguienteprimo(void)
{
  int esprimo, i;

  do {
    ultimoprimo++;
    esprimo = 1;
    for (i=2; i<ultimoprimo/2; i++)
      if (ultimoprimo % i == 0) {
        esprimo = 0;
        break;
      }
  } while (!esprimo);
  return ultimoprimo;
}

int main(void)
{
  int i;
```

```c

  printf("Los␣10␣primeros␣números␣primos\n");
  for (i=0; i<10; i++)
    printf("%d\n", siguienteprimo());
  return 0;
}
```

No hay problema con que las variables locales a una función sean vectores. Su contenido se almacena siempre en la pila. Este programa, por ejemplo, cuenta la cantidad de números primos entre 1 y el valor que se le indique (siempre que no supere cierta constante N) con la ayuda de la criba de Eratóstenes. El vector con el que se efectúa la criba es una variable local a la función que efectúa el conteo:

eratostenes_1.c — **eratostenes.c**

```c
#include <stdio.h>

#define N 10

int cuenta_primos(int n) //Cuenta el número de primos entre 1 y n.
{
  char criba[N];
  int i, j, numprimos;

  /* Comprobemos que el argumento es válido */
  if (n >= N)
    return -1; // Devolvemos −1 si no es válido.

  /* Inicialización */
  criba[0] = 0;
  for (i=1; i<n; i++)
    criba[i] = 1;

  /* Criba de Eratóstenes */
  for (i=2; i<n; i++)
    if (criba[i])
      for (j=2; i*j<n; j++)
        criba[i*j] = 0;

  /* Conteo de primos */
  numprimos = 0;
  for (i=0; i<n; i++)
    if (criba[i])
      numprimos++;

  return numprimos;
}


int main(void)
{
  int hasta, cantidad;

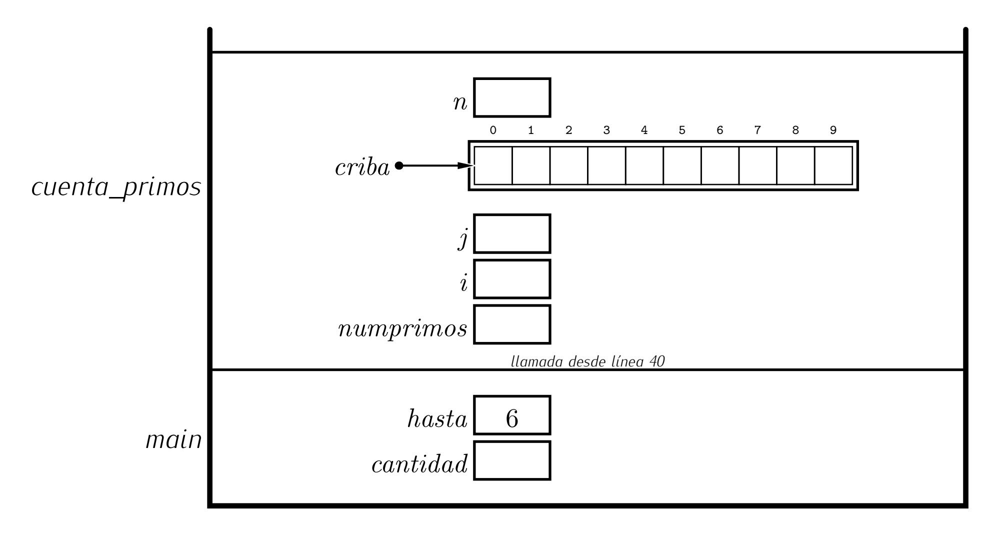
  printf("Introduce␣un␣valor:"); scanf("%d", &hasta);
  cantidad = cuenta_primos(hasta);
  if (cantidad == -1)
    printf("Imposible.␣El␣mayor␣valor␣permitido␣es␣%d.\n", N-1);
  else
    printf("Primos␣entre␣1␣y␣%d:␣%d\n", hasta, cantidad);
  return 0;
```

```
46 }
```

Cuando el programa inicia su ejecución, se crea una trama de activación en la que se albergan las variables *hasta* y *cantidad*. Supongamos que cuando se solicita el valor de *hasta* el usuario introduce el valor 6. He aquí el aspecto de la memoria:



Se efectúa entonces (línea 40) la llamada a *cuenta_primos*, con lo que se crea una nueva trama de activación. En ella se reserva memoria para todas las variables locales de *cuenta_primos*:



Observa que el vector *criba* ocupa memoria en la propia trama de activación. Completa tú mismo el resto de acciones ejecutadas por el programa ayudándote de una traza de la pila de llamadas a función con gráficos como los mostrados.

3.5.3. Vectores de longitud variable

Te hemos dicho en el apartado 2.1 que los vectores han de tener talla conocida en tiempo de compilación. Es hora de matizar esa afirmación. Los vectores locales a una función pueden determinar su talla en tiempo de ejecución. Veamos un ejemplo:

eratostenes_2.c — eratostenes.c

```
#include <stdio.h>

int cuenta_primos(int n) //Cuenta el número de primos entre 1 y n.
{
  char criba[n];
  int i, j, numprimos;

  /* Inicialización */
  criba[0] = 0;
  for (i=1; i<n; i++)
    criba[i] = 1;

  /* Criba de Eratóstenes */
  for (i=2; i<n; i++)
```

```
    if (criba[i])
      for (j=2; i*j<n; j++)
        criba[i*j] = 0;

  /* Conteo de primos */
  numprimos = 0;
  for (i=0; i<n; i++)
    if (criba[i])
      numprimos++;

  return numprimos;
}


int main(void)
{
  int hasta, cantidad;

  printf("Introduce un valor:"); scanf("%d", &hasta);
  cantidad = cuenta_primos(hasta);
  printf("Primos entre 1 y %d: %d\n", hasta, cantidad);
  return 0;
}
```

Fíjate en cómo hemos definido el vector *criba*: la talla no es un valor constante, sino *n*, un parámetro cuyo valor es desconocido hasta el momento en que se ejecute la función. Esta es una característica de C99 y supone una mejora interesante del lenguaje.

3.5.4. Parámetros vectoriales: paso por referencia

Este programa ilustra el modo en que podemos declarar y pasar parámetros vectoriales a una función:

pasa_vector.c

```
#include <stdio.h>

#define TALLA 3

void incrementa(int a[])
{
  int i;

  for (i=0; i<TALLA; i++)
    a[i]++;
}

int main(void)
{
  int i, v[TALLA];


  printf("Al principio:\n");
  for (i=0; i<TALLA; i++) {
    v[i] = i;
    printf("%d: %d\n", i, v[i]);
  }
  incrementa(v);
  printf("Después de llamar a incrementa:\n");
```

```
25    for (i=0; i<TALLA; i++)
26      printf("%d:␣%d\n", i, v[i]);
27    return 0;
28  }
```

Fíjate en cómo se indica que el parámetro *a* es un vector de enteros: añadiendo un par de corchetes a su identificador. En la línea 23 pasamos a *incrementa* el vector *v*. ¿Qué ocurre cuando modificamos componentes del parámetro vectorial *a* en la línea 10?

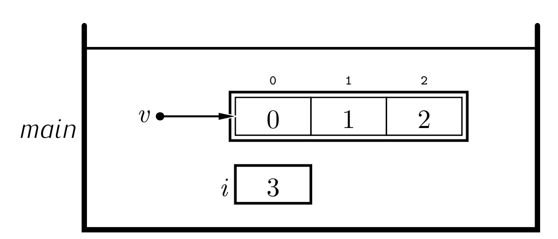
Si ejecutamos el programa obtenemos el siguiente texto en pantalla:

```
Al principio:
0: 0
1: 1
2: 2
Después de llamar a incrementa:
0: 1
1: 2
2: 3
```

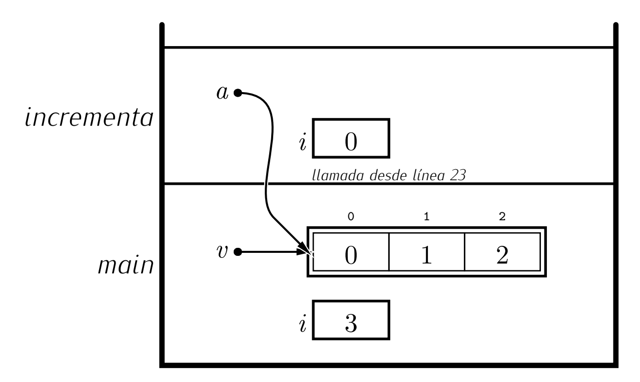
¡El contenido de *v* se ha modificado! Ocurre lo mismo que ocurría en Python: los vectores sí modifican su contenido cuando se altera el contenido del respectivo parámetro en las llamadas a función.

Cuando se pasa un parámetro vectorial a una función no se efectúa una copia de su contenido en la pila: sólo se copia la referencia a la posición de memoria en la que empieza el vector. ¿Por qué? Por eficiencia: no es infrecuente que los programas manejen vectores de tamaño considerable; copiarlos cada vez en la pila supondría invertir una cantidad de tiempo que, para vectores de tamaño medio o grande, podría ralentizar drásticamente la ejecución del programa. La aproximación adoptada por C hace que sólo sea necesario copiar en la pila 4 bytes, que es lo que ocupa una dirección de memoria. Y no importa cuán grande o pequeño sea un vector: la dirección de su primer valor siempre ocupa 4 bytes.

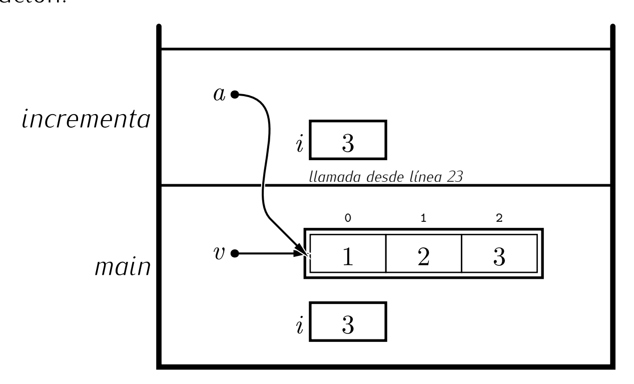
Veamos gráficamente, pues, qué ocurre en diferentes instantes de la ejecución del programa. Justo antes de ejecutar la línea 23 tenemos esta disposición de elementos en memoria:



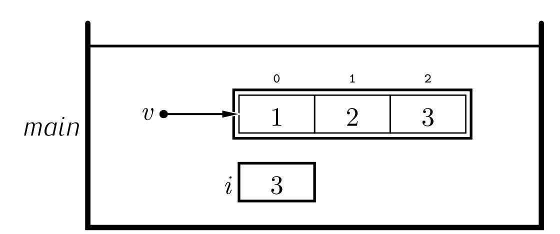
En el momento de ejecutar la línea 10 por primera vez, en la función *incrementa*, la memoria presenta este aspecto:



¿Ves? El parámetro *a* apunta a *v*. Los cambios sobre elementos del vector *a* que tienen lugar al ejecutar la línea 10 tienen efecto sobre los correspondientes elementos de *v*, así que *v* refleja los cambios que experimenta *a*. Tras ejecutar el bucle de *incrementa*, tenemos esta situación:



Y una vez ha finalizado la ejecución de *incrementa*, ésta otra:



¿Y qué ocurre cuando el vector es una variable global? Pues básicamente lo mismo: las referencias no tienen por qué ser direcciones de memoria de la pila. Este programa es básicamente idéntico al anterior, sólo que *v* es ahora una variable global:

pasa_vector_1.c — pasa_vector.c

```c
#include <stdio.h>

#define TALLA 3

int v[TALLA];

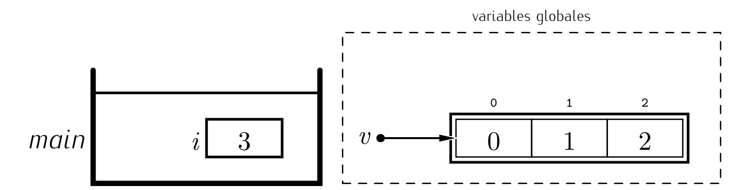
void incrementa(int a[])
{
  int i;

  for (i=0; i<TALLA; i++)
    a[i]++;
}

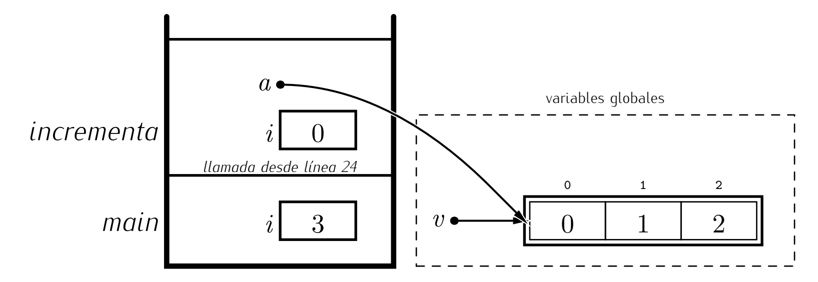
int main(void)
{
  int i;

  printf("Al principio:\n");
  for (i=0; i<TALLA; i++) {
    v[i] = i;
    printf("%d: %d\n", i, v[i]);
  }
  incrementa(v);
  printf("Después de llamar a incrementa:\n");
  for (i=0; i<TALLA; i++)
    printf("%d: %d\n", i, v[i]);
  return 0;
}
```

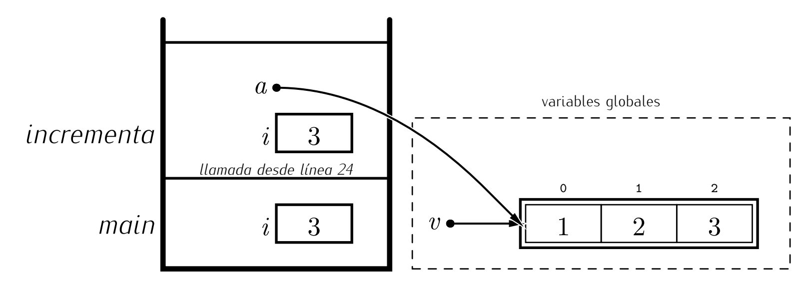
Analicemos qué ocurre en diferentes instantes de la ejecución del programa. Justo antes de ejecutar la línea 24, existen las variables locales a *main* y las variables globales:



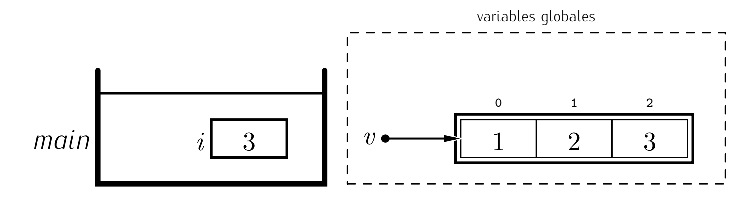
Al llamar a *incrementa* se suministra un puntero a la zona de memoria de variables globales, pero no hay problema alguno: el parámetro *a* es un puntero que apunta a esa dirección.



Los cambios al contenido de *a* se manifiestan en *v*:



Y una vez ha finalizado la ejecución de *incrementa*, el contenido de *v* queda modificado:



. . . . EJERCICIOS . . . .

▶ **170** Diseña un programa C que manipule polinomios de grado menor o igual que 10. Un polinomio se representará con un vector de **float** de tamaño 11. Si *p* es un vector que representa un polinomio, *p*[*i*] es el coeficiente del término de grado *i*. Diseña un procedimiento *suma* con el siguiente perfil:

void *suma*(**float** *p*[], **float** *q*[], **float** *r*[])

El procedimiento modificará *r* para que contenga el resultado de sumar los polinomios *p* y *q*.

▶ **171** Diseña una función que, dada una cadena y un carácter, diga cuántas veces aparece el carácter en la cadena.

. . . .

Hemos visto cómo pasar vectores a funciones. Has de ser consciente de que no hay forma de saber cuántos elementos tiene el vector dentro de una función: fíjate en que no se indica cuántos elementos tiene un parámetro vectorial. Si deseas utilizar el valor de la talla de un vector tienes dos posibilidades:

1. saberlo de antemano,

2. o proporcionarlo como parámetro adicional.

Estudiemos la primera alternativa. Fíjate en este fragmento de programa:

pasa_vector_talla.c

```
#include <stdio.h>

#define TALLA1 20
#define TALLA2 10

void inicializa(int z[])
{
  int i;

  for (i=0; i<TALLA1; i++)
    z[i] = 0;
}

void imprime(int z[])
{
  int i;

  for (i=0; i<TALLA1; i++)
    printf("%d ", z[i]);
  printf("\n");
}

int main(void)
{
  int x[TALLA1];
  int y[TALLA2];

  inicializa(x);
  inicializa(y); // ¡Ojo!

  imprime(x);
  imprime(y);   // ¡Ojo!

  return 0;
}
```

Siguiendo esta aproximación, la función *inicializa* sólo se puede utilizar con vectores de **int** de talla TALLA1, como *x*. No puedes llamar a *inicializa* con *y*: si lo haces (¡y C te deja hacerlo!) cometerás un error de acceso a memoria que no te está reservada, pues el bucle recorre TALLA1 componentes, aunque *y* sólo tenga TALLA2. Ese error puede abortar la ejecución del programa o, peor aún, no haciéndolo pero alterando la memoria de algún modo indefinido.

Este es el resultado obtenido en un ordenador concreto:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

El programa no ha abortado su ejecución, pero ha mostrado 20 valores del vector *y*, que sólo tiene 10.

¿Cómo podemos diseñar una función que pueda trabajar tanto con el vector *x* como con el vector *y*? Siguiendo la segunda aproximación propuesta, es decir, pasando como parámetro adicional la talla del vector en cuestión:

La dualidad vector/puntero, el paso de vectores y el paso por referencia

Al pasar un vector a una función pasamos una dirección de memoria: el inicio de la zona reservada para el vector. Cuando pasamos una variable escalar por referencia también pasamos una dirección de memoria: aquella en la que empieza la zona reservada para el valor escalar. ¿Qué diferencia hay entre una y otra dirección? Ninguna: un puntero siempre es un puntero. Fíjate en este programa:

dualidad.c   **dualidad.c**

```c
#include <stdio.h>

#define TALLA 10

void procedimiento(int *a, int b[])
{
  printf("%22d␣%6d\n", *a, b[0]);
  printf("%22d␣%6d\n", a[0], *b); // ¡Ojo!
}

int main(void)
{
  int x[TALLA], i, y = 10;

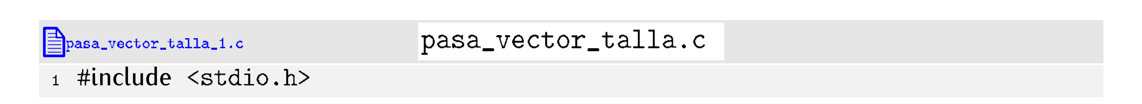
  for (i=0; i<TALLA; i++) x[i] = i + 1;
  printf("1)␣procedimiento(␣␣␣&y,␣␣␣␣␣x):\n");
  procedimiento(&y, x);
  printf("2)␣procedimiento(␣␣␣␣x,␣␣␣␣&y):\n");
  procedimiento(x, &y);
  printf("3)␣procedimiento(&x[0],␣&x[1]):\n");
  procedimiento(&x[0], &x[1]);

  return 0;
}
```

Esta es la salida resultante de su ejecución:

```
1) procedimiento(   &y,     x):
                    10      1
                    10      1
2) procedimiento(    x,    &y):
                     1     10
                     1     10
3) procedimiento(&x[0], &x[1]):
                     1      2
                     1      2
```

Observa qué ha ocurrido: en *procedimiento* se puede usar *a* y *b* como si fueran vectores o escalares pasados por referencia. Y podemos pasar a *procedimiento* tanto la dirección de un vector de **int**s como la de un escalar de tipo **int**. La conclusión es clara: «**int** * *a*» e «**int** *b*[]» son sinónimos cuando se declara un parámetro, pues en ambos casos se describen punteros a direcciones de memoria en las que residen sendos valores enteros (o donde empieza una serie de valores enteros). Aunque sean expresiones sinónimas y, por tanto, intercambiables, interesa que las uses «correctamente», pues así mejorará la legibilidad de tus programas: usa **int** * cuando quieras pasar la dirección de *un entero* y **int** [] cuando quieras pasar la dirección de *un vector de enteros*.


pasa_vector_talla_1.c   **pasa_vector_talla.c**

```c
#include <stdio.h>
```

```c

#define TALLA1 20
#define TALLA2 10

void inicializa(int z[], int talla)
{
  int i;

  for (i=0; i<talla; i++)
    z[i] = 0;
}

void imprime(int z[], int talla)
{
  int i;

  for (i=0; i<talla; i++)
    printf("%d␣", z[i]);
  printf("\n");
}


int main(void)
{
  int x[TALLA1];
  int y[TALLA2];

  inicializa(x, TALLA1);
  inicializa(y, TALLA2);

  imprime(x, TALLA1);
  imprime(y, TALLA2);

  return 0;
}
```

Ahora puedes llamar a la función *inicializa* con *inicializa*(*x*, TALLA1) o *inicializa*(*y*, TALLA2). Lo mismo ocurre con *imprime*. El parámetro *talla* toma el valor apropiado en cada caso porque tú se lo estás pasando explícitamente.

Éste es el resultado de ejecutar el programa ahora:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Correcto.

. . . . EJERCICIOS . . . .

▶ **172** Diseña un procedimiento *ordena* que ordene un vector de enteros. El procedimiento recibirá como parámetros un vector de enteros y un entero que indique el tamaño del vector.

▶ **173** Diseña una función que devuelva el máximo de un vector de enteros. El tamaño del vector se suministrará como parámetro adicional.

▶ **174** Diseña una función que diga si un vector de enteros es o no es palíndromo (devolviendo 1 o 0, respectivamente). El tamaño del vector se suministrará como parámetro adicional.

▶ **175** Diseña una función que reciba dos vectores de enteros de idéntica talla y diga si son iguales o no. El tamaño de los dos vectores se suministrará como parámetro adicional.

▶ **176** Diseña un *procedimiento* que reciba un vector de enteros y muestre todos sus componentes en pantalla. Cada componente se representará separado del siguiente con una coma. El último elemento irá seguido de un salto de línea. La talla del vector se indicará con un parámetro adicional.

▶ **177** Diseña un *procedimiento* que reciba un vector de **float** y muestre todos sus componentes en pantalla. Cada componente se representará separado del siguiente con una coma. Cada 6 componentes aparecerá un salto de línea. La talla del vector se indicará con un parámetro adicional.

3.5.5. Parámetros escalares: paso por referencia mediante punteros

C permite modificar el valor de variables escalares en una función recurriendo a sus direcciones de memoria. Analicemos el siguiente ejemplo:

referencia_local.c

```
1  #include <stdio.h>
2
3  void incrementa(int * a)
4  {
5    *a += 1;
6  }
7
8  int main(void)
9  {
10   int b;
11
12   b = 1;
13   printf("Al␣principio␣b␣vale␣%d\n", b);
14   incrementa(&b);
15   printf("Y␣al␣final␣vale␣%d\n", b);
16   return 0;
17 }
```

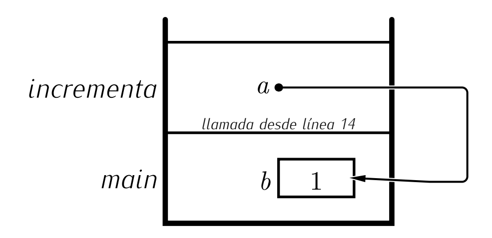
Al ejecutarlo, aparece en pantalla el siguiente texto:

```
Al principio b vale 1
Y al final vale 2
```

Efectivamente, *b* ha modificado su valor tras la llamada a *incrementa*. Observa la forma en que se ha declarado el único parámetro de *incrementa*: **int** * *a*. O sea, *a* es del tipo **int** *. Un tipo de la forma «**tipo** *» significa «puntero a valor de tipo **tipo**». Tenemos, por tanto, que *a* es un «puntero a entero». No le pasamos a la función *el valor* de un entero, sino *el valor de la dirección* de memoria en la que se encuentra un entero.

Fíjate ahora en cómo pasamos el argumento en la llamada a *incrementa* de la línea 14, que es de la forma *incrementa*(&*b*). Estamos pasando *la dirección de memoria* de *b* (que es lo que proporciona el operador &) y no el valor de *b*. Todo correcto, ya que hemos dicho que la función espera la dirección de memoria de un entero.

Al principio de la ejecución de *incrementa* tendremos esta situación:



El parámetro *a* es un puntero que apunta a *b*. Fíjate ahora en la sentencia que incrementa el valor apuntado por *a* (línea 5):

**a* += 1;

El asterisco que precede a *a* no indica «multiplicación». Ese asterisco es un operador unario que hace justo lo contrario que el operador &: dada una dirección de memoria, accede al valor de la variable apuntada. (Recuerda que el operador & obtenía la dirección de memoria de una variable.) O sea, C interpreta **a* como accede a la variable apuntada por *a*, que es *b*, así que **a* += 1 equivale a *b* += 1 e incrementa el contenido de la variable *b*.

> ***El & de los parámetros de* scanf**
>
> Ahora ya puedes entender bien por qué las variables escalares que suministramos a *scanf* para leer su valor por teclado van precedidas por el operador &: como *scanf* debe modificar su valor, ha de saber en qué dirección de memoria residen. No ocurre lo mismo cuando vamos a leer una cadena, pero eso es porque el identificador de la variable ya es, en ese caso, una dirección de memoria.

¿Qué pasaría si en lugar de **a* += 1 hubiésemos escrito *a* += 1? Se hubiera incrementado la dirección de memoria a la que apunta el puntero, nada más.

¿Y si hubiésemos escrito *a*++? Lo mismo: hubiésemos incrementado el valor de la dirección almacenada en *a*. ¿Y **a*++?, ¿funcionaría? A primera vista diríamos que sí, pero no funciona como esperamos. El operador ++ postfijo tiene mayor nivel de precedencia que el operador unario *, así que **a*++ (post)incrementa la dirección *a* y accede a su contenido, por ese órden. Nuevamente habríamos incrementado el valor de la dirección de memoria, y no su contenido. Si quieres usar operadores de incremento/decremento, tendrás que utilizar paréntesis para que los operadores se apliquen en el orden deseado: (**a*)++.

Naturalmente, no sólo puedes acceder así a variables locales, también las variables globales son accesibles mediante punteros:

referencia_global.c

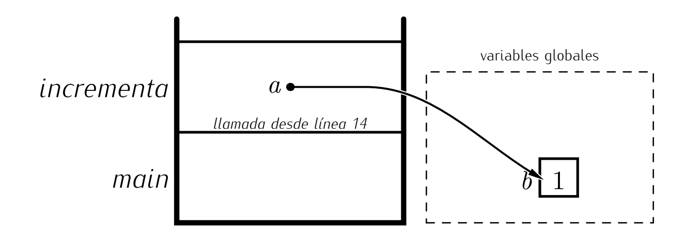
```
#include <stdio.h>

int b; // Variable global.

void incrementa(int * a)
{
  *a += 1;
}

int main(void)
{
  b = 1;
  printf("Al principio b vale %d\n", b);
  incrementa(&b);
  printf("Y al final vale %d\n", b);
  return 0;
}
```

El aspecto de la memoria cuando empieza a ejecutarse la función *incrementa* es éste:



. . . . EJERCICIOS . . . .

▶ **178** Diseña un procedimiento que modifique el valor del parámetro de tipo **float** para que valga la inversa de su valor cuando éste sea distinto de cero. Si el número es cero, el procedimiento dejará intacto el valor del parámetro.

Si *a* vale 2.0, por ejemplo, *inversa*(&*a*) hará que *a* valga 0.5.

▶ **179** Diseña un procedimiento que intercambie el valor de dos números enteros.

Si *a* y *b* valen 1 y 2, respectivamente, la llamada *intercambia*(&*a*, &*b*) hará que *a* pase a valer 2 y *b* pase a valer 1.

▶ **180** Diseña un procedimiento que intercambie el valor de dos números **float**.

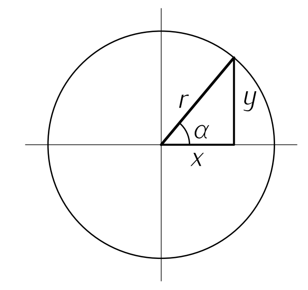
▶ **181** Diseña un procedimiento que asigne a todos los elementos de un vector de enteros un valor determinado. El procedimiento recibirá tres datos: el vector, su número de elementos y el valor que que asignamos a todos los elementos del vector.

▶ **182** Diseña un procedimiento que intercambie el contenido completo de dos vectores de enteros de igual talla. La talla se debe suministrar como parámetro.

▶ **183** Diseña un procedimiento que asigne a un entero la suma de los elementos de un vector de enteros. Tanto el entero (su dirección) como el vector se suministrarán como parámetros.

. . . .

Un uso habitual del paso de parámetros por referencia es la devolución de más de un valor como resultado de la ejecución de una función. Veámoslo con un ejemplo. Diseñemos una función que, dados un ángulo α (en radianes) y un radio r, calcule el valor de $x = r\cos(\alpha)$ e $y = r\sin(\alpha)$:



No podemos diseñar una función que devuelva los dos valores. Hemos de diseñar un procedimiento que devuelva los valores resultantes como parámetros pasados por referencia:

paso_por_referencia.c

```
#include <stdio.h>
#include <math.h>

void calcula_xy(float alfa, float radio, float * x , float * y )
{
  *x = radio * cos(alfa);
  *y = radio * sin(alfa);
}
```

En C sólo hay paso por valor

Este apartado intenta que aprendas a distinguir el paso de parámetros por valor y por referencia. ¡Pero la realidad es que C sólo tiene paso de parámetros por valor! Cuando pasas una referencia, estás pasando explícitamente una dirección de memoria gracias al operador `&`, y lo que hace C es copiar dicha dirección en la pila, es decir, pasa por valor una dirección para *simular* el paso de parámetros por referencia. La extraña forma de pasar el parámetro hace que tengas que usar el operador `*` cada vez que deseas acceder a él en el cuerpo de la función.

En otros lenguajes, como Pascal, es posible indicar que un parámetro se pasa por referencia sin que tengas que usar un operador (equivalente a) `&` al efectuar el paso o un operador (equivalente a) `*` cuando usas el parámetro en el cuerpo de la función. Por ejemplo, este programa Pascal incluye un procedimiento que modifica el valor de su parámetro:

```
program referencia;

var b : integer;

procedure incrementa (var a : integer);
begin
  a := a + 1;
end;

begin (* programa principal *)
  b := 1;
  writeln('b valía ', b);
  incrementa(b);
  writeln('b vale ', b)
end.
```

C++ es una extensión de C que permite el paso de parámetros por referencia. Usa para ello el carácter `&` en la declaración del parámetro:

```
#include <stdio.h>

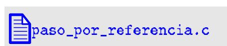
void incrementa(int & a)
{
  a += 1;
}

int main(void)
{
  int b;

  b = 1;
  printf("Al␣principio␣b␣vale␣%d\n", b);
  incrementa(b);
  printf("Y␣al␣final␣vale␣%d\n", b);
  return 0;
}
```

(Aunque no venga a cuento, observa lo diferente que es C de Pascal (y aun así, lo semejante que es) y cómo el programa C++ presenta un aspecto muy semejante a uno equivalente escrito en C.)

¿Y cómo llamamos al procedimiento? Aquí tienes un ejemplo de uso:



paso_por_referencia.c

```
  ⋮
}

int main(void)
{
  float r, angulo, horizontal, vertical;

  printf("Introduce el ángulo (en radianes): "); scanf("%f", &angulo);
  printf("Introduce el radio: "); scanf("%f", &r);
  calcula_xy(angulo, r, &horizontal, &vertical);
  printf("Resultado: (%f, %f)\n", horizontal, vertical);

  return 0;
}
```

¿Ves? Las variables *horizontal* y *vertical* no se inicializan en *main*: reciben valores como resultado de la llamada a *calcula_xy*.

. . . . EJERCICIOS . . . .

▶ **184** Diseña una función que calcule la inversa de *calcula_xy*, es decir, que obtenga el valor del radio y del ángulo a partir de *x* e *y*.

▶ **185** Diseña una función que reciba dos números enteros *a* y *b* y devuelva, simultáneamente, el menor y el mayor de ambos. La función tendrá esta cabecera:

```
void minimax (int a, int b, int * min, int * max)
```

▶ **186** Diseña una función que reciba un vector de enteros, su talla y un valor de tipo entero al que denominamos *buscado*. La función devolverá (mediante return) el valor 1 si *buscado* tiene el mismo valor que algún elemento del vector y 0 en caso contrario. La función devolverá, además, la distancia entre *buscado* y el elemento más próximo a él.

La cabecera de la función ha de ser similar a ésta:

```
int busca(int vector[], int talla, int buscado, int * distancia)
```

Te ponemos un par de ejemplos para que veas qué debe hacer la función.

```
#include <stdio.h>

#define TALLA 6

// Define aquí la función
...

int main(void)
{
  int v[TALLA], distancia, encontrado, buscado, i;

  for (i=0; i<TALLA; i++) {
    printf("Introduce el elemento %d: ", i);
    scanf("%d", &v[i]);
  }

  printf("¿Qué valor busco?: ");
  scanf("%d", &buscado);

  encontrado = busca(v, TALLA, buscado, &distancia);
  if (encontrado)
    printf("Encontré el valor %d.\n", buscado);
  else
```

```
    printf("No␣está.␣El␣más␣próximo␣está␣a␣distancia␣%d.\n", distancia);

  printf("¿Qué␣valor␣busco␣ahora?:␣");
  scanf("%d", &buscado);

  encontrado = busca(v, TALLA, buscado, &distancia);
  if (encontrado)
    printf("Encontré␣el␣valor␣%d.\n", buscado);
  else
    printf("No␣está.␣El␣más␣próximo␣está␣a␣distancia␣%d.\n", distancia);

  return 0;
}
```

Al ejecutar el programa obtenemos esta salida por pantalla:

```
Introduce el elemento: 0 ↲
Introduce el elemento: 5 ↲
Introduce el elemento: 10 ↲
Introduce el elemento: 15 ↲
Introduce el elemento: 20 ↲
Introduce el elemento: 25 ↲
¿Qué valor busco?: 5 ↲
Encontré el valor 5.
¿Qué valor busco ahora?: 17 ↲
No está. El elemento más próximo está a distancia 2.
```

▶ 187 Modifica la función del ejercicio anterior para que, además de la distancia al elemento más próximo, devuelva el valor del elemento más próximo.

▶ 188 Modifica la función del ejercicio anterior para que, además de la distancia al elemento más próximo y el elemento más próximo, devuelva el valor de su índice.

. . . .

3.5.6. Paso de registros a funciones

No sólo puedes pasar escalares y vectores como argumentos, también puedes pasar registros. El paso de registros es por valor, o sea, copiando el contenido en la pila, a menos que tú mismo pases un puntero a su dirección de memoria.

Este programa, por ejemplo, define un tipo de datos para representar puntos en un espacio de tres dimensiones y una función que calcula la distancia de un punto al origen:

```
#include <stdio.h>
#include <math.h>

struct Punto {
  float x, y, z;
};

float distancia(struct Punto p)
{
  return sqrt( p.x*p.x + p.y*p.y + p.z*p.z );
}

int main(void)
{
  struct Punto pto;
```

```

    pto.x = 1;
    pto.y = 1;
    pto.z = 1;

    printf("Distancia al origen: %f\n", distancia(pto));
    return 0;
}
```

Al pasar un registro a la función, C copia en la pila cada uno de los valores de sus campos. Ten en cuenta que una variable de tipo **struct** *Punto* ocupa 24 bytes (contiene 3 valores de tipo **float**). Variables de otros tipos registro que definas pueden ocupar cientos o incluso miles de bytes, así que ve con cuidado: llamar a una función pasando registros por valor puede resultar ineficiente. Por cierto, no es tan extraño que un registro ocupe cientos de bytes: uno o más de sus campos podría ser un vector. También en ese caso se estaría copiando su contenido íntegro en la pila.

Eso sí, como estás pasando una copia, las modificaciones del valor de un campo en el cuerpo de la función no tendrán efectos perceptibles fuera de la función.

Como te hemos anticipado, también puedes pasar registros por referencia. En tal caso sólo se estará copiando en la pila la dirección de memoria en la que empieza el registro (y eso son 4 bytes), mida lo que mida éste. Se trata, pues, de un paso de parámetros más eficiente. Eso sí, has de tener en cuenta que los cambios que efectúes a cualquier campo del parámetro se reflejarán en el campo correspondiente de la variable que suministraste como argumento.

Esta función, por ejemplo, define dos parámetros: uno que se pasa por referencia y otro que se pasa por valor. La función traslada un punto *p* en el espacio (modificando los campos del punto original) de acuerdo con el vector de desplazamiento que se indica con otro punto (*traslacion*):

```
void traslada(struct Punto * p, struct Punto traslacion)
{
    (*p).x += traslacion.x;
    (*p).y += traslacion.y;
    (*p).z += traslacion.z;
}
```

Observa cómo hemos accedido a los campos de *p*. Ahora *p* es una dirección de memoria (es de tipo **struct** *Punto* `*`), y `*`*p* es la variable apuntada por *p* (y por tanto, es de tipo **struct** *Punto*). El campo *x* es accedido con `(*p).x`: *primero* se accede al contenido de la dirección de memoria apuntada por *p*, y *luego* al campo *x* del registro `*`*p*, de ahí que usemos paréntesis.

Es tan frecuente la notación `(*p).x` que existe una forma compacta equivalente:

```
void traslada(struct Punto * p, struct Punto traslacion)
{
    p->x += traslacion.x;
    p->y += traslacion.y;
    p->z += traslacion.z;
}
```

La forma `p->x` es absolutamente equivalente a `(*p).x`.

Recuerda, pues, que dentro de una función se accede a los campos de forma distinta según se pase un valor por copia o por referencia:

1. con el operador punto, como en *traslacion.x*, si la variable se ha pasado por valor;

2. con el operador «flecha», como en `p->x`, si la variable se ha pasado por referencia (equivalentemente, puedes usar la notación `(*p).x`).

Acabemos este apartado mostrando una rutina que pide al usuario que introduzca las coordenadas de un punto:

```
void lee_punto(struct Punto * p)
{
  printf("x: "); scanf("%f", &p->x);
  printf("y: "); scanf("%f", &p->y);
  printf("z: "); scanf("%f", &p->z);
}
```

. . . . EJERCICIOS . . . .

▶ **189** Este ejercicio y los siguientes de este bloque tienen por objeto construir una serie de funciones que permitan efectuar transformaciones afines sobre puntos en el plano. Los puntos serán variables de tipo **struct** *Punto*, que definimos así:

```
struct Punto {
  float x, y;
};
```

Diseña un procedimiento *muestra_punto* que muestre por pantalla un punto. Un punto *p* tal que *p*.*x* vale 2.0 y *p*.*y* vale 0.2 se mostrará en pantalla así: (2.000000, 0.200000). El procedimiento *muestra_punto* recibirá un punto *por valor*.

Diseña a continuación un procedimiento que permita leer por teclado un punto. El procedimiento recibirá *por referencia* el punto en el que se almacenarán los valores leídos.

▶ **190** La operación de traslación permite desplazar un punto de coordenadas (x, y) a $(x + a, y + b)$, siendo el desplazamiento (a, b) un vector (que representamos con otro punto). Implementa una función que reciba dos parámetros de tipo punto y modifique el primero de modo que se traslade lo que indique el vector.

▶ **191** La operación de escalado transforma un punto (x, y) en otro (ax, ay), donde a es un factor de escala (real). Implementa una función que escale un punto de acuerdo con el factor de escala a que se suministre como parámetro (un **float**).

▶ **192** Si rotamos un punto (x, y) una cantidad de θ *radianes* alrededor del origen, obtenemos el punto

$$(x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta).$$

Define una función que rote un punto la cantidad de *grados* que se especifique.

▶ **193** La rotación de un punto (x, y) una cantidad de θ radianes alrededor de un punto (a, b) se puede efectuar con una traslación con el vector $(-a, -b)$, una rotación de θ radianes con respecto al origen y una nueva traslación con el vector (a, b). Diseña una función que permita trasladar un punto un número dado de grados alrededor de otro punto.

▶ **194** Diseña una función que diga si dos puntos son iguales.

▶ **195** Hemos definido un tipo registro para representar complejos así:

```
struct Complejo {
  float real;
  float imag;
};
```

Diseña e implementa los siguientes procedimientos para su manipulación:

- leer un complejo de teclado;
- mostrar un complejo por pantalla;

- el módulo de un complejo ($|a + bi| = \sqrt{a^2 + b^2}$);
- el opuesto de un complejo ($-(a + bi) = -a - bi$);
- el conjugado de un complejo ($\overline{a + bi} = a - bi$);
- la suma de dos complejos ($(a + bi) + (c + di) = (a + c) + (b + d)i$);
- la diferencia de dos complejos ($(a + bi) - (c + di) = (a - c) + (b - d)i$);
- el producto de dos complejos ($(a + bi) \cdot (c + di) = (ac - bd) + (ad + bc)i$);
- la división de dos complejos ($\frac{a+bi}{c+di} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i$).

▶ 196 Define un tipo registro y una serie de funciones para representar y manipular fechas. Una fecha consta de un día, un mes y un año. Debes implementar funciones que permitan:

- mostrar una fecha por pantalla con formato *dd/mm/aaaa* (por ejemplo, el 7 de junio de 2001 se muestra así: 07/06/2001);
- mostrar una fecha por pantalla como texto (por ejemplo, el 7 de junio de 2001 se muestra así: `7 de junio de 2001`);
- leer una fecha por teclado;
- averiguar si una fecha cae en año bisiesto;
- averiguar si una fecha es anterior, igual o posterior a otra, devolviendo los valores −1, 0 o 1 respectivamente,
- comprobar si una fecha existe (por ejemplo, el 29 de febrero de 2002 no existe):
- calcular la diferencia de días entre dos fechas.

3.5.7. Paso de matrices y otros vectores multidimensionales

El paso de vectores multidimensionales no es una simple extensión del paso de vectores unidimensionales. Veamos. Aquí tienes un programa incorrecto en el que se define una función que recibe una matriz y devuelve su elemento máximo:

pasa_matriz_mal.c ↯ pasa_matriz_mal.c ↯

```
#include <stdio.h>

#define TALLA 3

int maximo(int a[][])
{
  int i, j, m;

  m = a[0][0];
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      if (a[i][j] > m)
        m = a[i][j];

  return m;
}
```

```c

int main(void)
{
  int matriz[TALLA][TALLA];
  int i, j;

  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      matriz[i][j] = (i*j) % TALLA;

  printf("El máximo es %d\n", maximo(matriz));
  return 0;
}
```

El compilador no acepta ese programa. ¿Por qué? Fíjate en la declaración del parámetro. ¿Qué hay de malo? C no puede resolver los accesos de la forma *a*[*i*][*j*]. Si recuerdas, *a*[*i*][*j*] significa «accede a la celda cuya dirección se obtiene sumando a la dirección *a* el valor *i* ∗ COLUMNAS + j», donde COLUMNAS es el número de columnas de la matriz *a* (en nuestro caso, sería TALLA). Pero, ¿cómo sabe la función cuántas columnas tiene *a*? ¡No hay forma de saberlo viendo una definición del estilo **int** *a*[][]!

La versión correcta del programa debe indicar explícitamente cuántas columnas tiene la matriz. Hela aquí:

pasa_matriz.c **pasa_matriz.c**

```c
#include <stdio.h>

#define TALLA 3

int maximo(int a[][TALLA])
{
  int i, j, m;

  m = a[0][0];
  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      if (a[i][j] > m)
        m = a[i][j];

  return m;
}

int main(void)
{
  int matriz[TALLA][TALLA];
  int i, j;

  for (i=0; i<TALLA; i++)
    for (j=0; j<TALLA; j++)
      matriz[i][j] = (i*j) % TALLA;

  printf("El máximo es %d\n", maximo(matriz));
  return 0;
}
```

No ha sido necesario indicar cuántas filas tiene la matriz (aunque somos libres de hacerlo). La razón es sencilla: el número de filas no hace falta para calcular la dirección en la que reside el valor *a*[*i*][*j*].

Así pues, en general, es necesario indicar explícitamente el tamaño de cada una de las dimensiones del vector, excepto el de la primera (que puedes declarar o no, a voluntad).

Sólo así obtiene C información suficiente para calcular los accesos a elementos del vector en el cuerpo de la función.

Una consecuencia de esta restricción es que no podremos definir funciones capaces de trabajar con matrices de tamaño arbitrario. Siempre hemos de definir explícitamente el tamaño de cada dimensión excepto de la primera. Habrá una forma de superar este inconveniente, pero tendremos que esperar al siguiente capítulo para poder estudiarla.

. . . . EJERCICIOS . . . .

▶ **197** Vamos a diseñar un programa capaz de jugar al tres en raya. El tablero se representará con una matriz de 3×3. Las casillas serán caracteres. El espacio en blanco representará una casilla vacía; el carácter `'o'` representará una casilla ocupada con un círculo y el carácter `'x'` representará una casilla marcada con una cruz.

- Diseña una función que muestre por pantalla el tablero.
- Diseña una función que detecte si el tablero está lleno.
- Diseña una función que detecte si algún jugador consiguió hacer tres en raya.
- Diseña una función que solicite al usuario la jugada de los círculos y modifique el tablero adecuadamente. La función debe detectar si la jugada es válida o no.
- Diseña una función que, dado un tablero, realice la jugada que corresponde a las cruces. En una primera versión, haz que el ordenador ponga la cruz en la primera casilla libre. Después, modifica la función para que el ordenador realice la jugada más inteligente.

Cuando hayas diseñado todas las funciones, monta un programa que las use y permita jugar al tres en raya contra el computador.

▶ **198** El juego de la vida se juega sobre una matriz cuyas celdas pueden estar vivas o muertas. La matriz se modifica a partir de su estado siguiendo unas sencilla reglas que tienen en cuenta los, como mucho, 8 vecinos de cada casilla:

- Si una celda viva está rodeada por 0 o 1 celdas vivas, muere de soledad.
- Si una celda viva está rodeada por 4 celdas vivas, muere por superpoblación.
- Si una celda viva está rodeada por 2 o 3 celdas vivas, sigue viva.
- Una celda muerta sólo resucita si está rodeada por 3 celdas vivas.

Diseña una función que reciba una matriz de 10×10 celdas en la que el valor 0 representa «celda muerta» y el valor 1 representa «celda viva». La función modificará la matriz de acuerdo con las reglas del juego de la vida. (Avisos: Necesitarás una matriz auxiliar. Las celdas de los bordes no tienen 8 vecinos, sino 3 o 5.)

A continuación, monta un programa que permita al usuario introducir una disposición inicial de celdas y ejecutar el juego de la vida durante n ciclos, siendo n un valor introducido por el usuario.

Aquí tienes un ejemplo de «partida» de 3 ciclos con una configuración inicial curiosa:

```
Configuración inicial:
__________
______xxx_
__________
__________
___xxx____
__xxx_____
__________
```

```
__________
__________
__________
Ciclos: 3
_______x__
_______x__
_______x__
____x_____
__x__x____
__x__x____
___x______
__________
__________
__________

__________
______xxx_
__________
__________
___xxx____
__xxx_____
__________
__________
__________
__________

_______x__
_______x__
_______x__
____x_____
__x__x____
__x__x____
___x______
__________
__________
__________
```

▶ 199 Implementa el juego del buscaminas. El juego del buscaminas se juega en un tablero de dimensiones dadas. Cada casilla del tablero puede contener una bomba o estar vacía. Las bombas se ubican aleatoriamente. El usuario debe descubrir todas las casillas que no contienen bomba. Con cada jugada, el usuario descubre una casilla (a partir de sus coordenadas, un par de letras). Si la casilla contiene una bomba, la partida finaliza con la derrota del usuario. Si la casilla está libre, el usuario es informado de cuántas bombas hay en las (como mucho) 8 casillas vecinas.

Este tablero representa, en un terminal, el estado actual de una partida sobre un tablero de 8×8:

```
  abcdefgh
a 00001___
b 00112___
c 222_____
d ________
e ____3___
f ________
g 1_111111
h __100000
```

Las casillas con un punto no han sido descubiertas aún. Las casillas con un número han sido descubiertas y sus casillas vecinas contienen tantas bombas como se indica en el

número. Por ejemplo, la casilla de coordenadas (`'e'`, `'e'`) tiene 3 bombas en la vecindad y la casilla de coordenadas (`'b'`, `'a'`), ninguna.

Implementa un programa que permita seleccionar el nivel de dificultad y, una vez escogido, genere un tablero y permita jugar con él al jugador.

Los niveles de dificultad son:

- fácil: tablero de 8×8 con 10 bombas.
- medio: tablero de 15×15 con 40 bombas.
- difícil: tablero de 20×20 con 100 bombas.

Debes diseñar funciones para desempeñar cada una de las acciones básicas de una partida:

- dado un tablero y las coordenadas de una casilla, indicar si contiene bomba o no,
- dado un tablero y las coordenadas de una casilla, devolver el número de bombas vecinas,
- dado un tablero y las coordenadas de una casilla, modificar el tablero para indicar que la casilla en cuestión ya ha sido descubierta,
- dado un tablero, mostrar su contenido en pantalla,
- etc.

3.5.8. Tipos de retorno válidos

Una función puede devolver valores de cualquier tipo escalar o de registros, pero no puede devolver vectores[3]. La razón es simple: la asignación funciona con valores escalares y registros, pero no con vectores.

Ya hemos visto cómo devolver valores escalares. A título ilustrativo te presentamos un ejemplo de definición de registro y definición de función que recibe como parámetros un punto (x, y) y un número y devuelve un nuevo punto cuyo valor es (ax, ay):

```
struct Punto {
  float x, y;
};

struct Punto escala(struct Punto p, float a)
{
  struct Punto q;

  q.x = a * p.x;
  q.y = a * p.y;

  return q;
}
```

Eso es todo... por el momento. Volveremos a la cuestión de si es posible devolver vectores cuando estudiemos la gestión de memoria dinámica.

EJERCICIOS

▶ **200** Vuelve a implementar las funciones de manipulación de puntos en el plano (ejercicios 189–194) para que no modifiquen el valor del registro **struct** *Punto* que se suministra como parámetro. En su lugar, devolverán el punto resultante como valor de retorno de la llamada a función.

[3] Al menos no hasta que sepamos más de la gestión de memoria dinámica

▶ 201 Implementa nuevamente las funciones del ejercicio 195, pero devolviendo un nuevo complejo con el resultado de operar con el complejo o complejos que se suministran como parámetros.

. . . .

3.5.9. Un ejercicio práctico: miniGalaxis

Pongamos en práctica lo aprendido diseñando una versión simplificada de un juego de rescate espacial (Galaxis)[4] al que denominaremos miniGalaxis.

MiniGalaxis se juega con un tablero de 9 filas y 20 columnas. En el tablero hay 5 náufragos espaciales y nuestro objetivo es descubrir dónde se encuentran. Contamos para ello con una sonda espacial que podemos activar en cualquier casilla del tablero. La sonda dispara una señal en las cuatro direcciones cardinales que es devuelta por unos dispositivos que llevan los náufragos. La sonda nos dice cuántos náufragos espaciales han respondido, pero no desde qué direcciones enviaron su señal de respuesta. Cuando activamos la sonda en las coordenadas exactas en las que se encuentra un naúfrago, lo damos por rescatado. Sólo disponemos de 20 sondas para efectuar el rescate, así que las hemos de emplear juiciosamente. De lo contrario, la muerte de inocentes pesará sobre nuestra conciencia.

Lo mejor será que te hagas una idea precisa del juego jugando. Al arrancar aparece esta información en pantalla:

```
  ABCDEFGHIJKLMNOPQRST
0 ++++++++++++++++++++
1 ++++++++++++++++++++
2 ++++++++++++++++++++
3 ++++++++++++++++++++
4 ++++++++++++++++++++
5 ++++++++++++++++++++
6 ++++++++++++++++++++
7 ++++++++++++++++++++
8 ++++++++++++++++++++
Hay 5 náufragos.
Dispones de 20 sondas.
Coordenadas:
```

El tablero se muestra como una serie de casillas. Arriba tienes letras para identificar las columnas y a la izquierda números para las filas. El ordenador nos informa de que aún quedan 5 náufragos por rescatar y que disponemos de 20 sondas. Se ha detenido mostrando el mensaje «Coordenadas:»: está esperando a que digamos en qué coordenadas lanzamos una sonda. El ordenador acepta una cadena que contenga un dígito y una letra (en cualquier orden) y la letra puede ser minúscula o mayúscula. Lancemos nuestra primera sonda: escribamos 5b y pulsemos la tecla de retorno de carro. He aquí el resultado:

```
Coordenadas: 5b ↲
  ABCDEFGHIJKLMNOPQRST
0 +.++++++++++++++++++
1 +.++++++++++++++++++
2 +.++++++++++++++++++
3 +.++++++++++++++++++
4 +.++++++++++++++++++
```

[4] El nombre y la descripción puede que te hagan concebir demasiadas esperanzas: se trata de un juego muy sencillito y falto de cualquier efecto especial. Galaxis fue concebido por Christian Franz y escrito para el Apple Macintosh. Más tarde, Eric Raymond lo reescribió para que fuera ejecutable en Unix.

```
5 .0..................
6 +.++++++++++++++++++
7 +.++++++++++++++++++
8 +.++++++++++++++++++
Hay 5 náufragos.
Dispones de 19 sondas.
Coordenadas:
```

El tablero se ha redibujado y muestra el resultado de lanzar la sonda. En la casilla de coordenadas `5b` aparece un cero: es el número de naúfragos que hemos detectado con la sonda. Mala suerte. Las casillas que ahora aparecen con un punto son las exploradas por la sonda. Ahora sabes que en ninguna de ellas hay un náufrago. Sigamos jugando: probemos con las coordenadas `3I`. Aquí tienes la respuesta del ordenador:

```
Coordenadas: 3I ↲
  ABCDEFGHIJKLMNOPQRST
0 +.++++++.+++++++++++
1 +.++++++.+++++++++++
2 +.++++++.+++++++++++
3 ........1...........
4 +.++++++.+++++++++++
5 .0..................
6 +.++++++.+++++++++++
7 +.++++++.+++++++++++
8 +.++++++.+++++++++++
Hay 5 náufragos.
Dispones de 18 sondas.
Coordenadas:
```

En la casilla de coordenadas `3I` aparece un uno: la sonda ha detectado la presencia de un náufrago en alguna de las 4 direcciones. Sigamos. Probemos en `0I`:

```
Coordenadas: i0 ↲
  ABCDEFGHIJKLMNOPQRST
0 ........2...........
1 +.++++++.+++++++++++
2 +.++++++.+++++++++++
3 ........1...........
4 +.++++++.+++++++++++
5 .0..................
6 +.++++++.+++++++++++
7 +.++++++.+++++++++++
8 +.++++++.+++++++++++
Hay 5 náufragos.
Dispones de 17 sondas.
Coordenadas:
```

Dos náufragos detectados. Parece probable que uno de ellos esté en la columna `I`. Lancemos otra sonda en esa columna. Probemos con 2`I`:

```
Coordenadas: 2I ↲
  ABCDEFGHIJKLMNOPQRST
0 ........2...........
1 +.++++++.+++++++++++
2 ........X...........
3 ........1...........
4 +.++++++.+++++++++++
5 .0..................
6 +.++++++.+++++++++++
```

```
7 +.++++++.+++++++++++
8 +.++++++.+++++++++++
Hay 4 náufragos.
Dispones de 16 sondas.
Coordenadas:
```

¡Bravo! Hemos encontrado a uno de los náufragos. En el tablero se muestra con una X. Ya sólo quedan 4.

Bueno. Con esta partida inacabada puedes hacerte una idea detallada del juego. Diseñemos el programa.

Empezamos por definir las estructuras de datos. La primera de ellas, el tablero de juego, que es una simple matriz de 9×20 casillas. Nos vendrá bien disponer de constantes que almacenen el número de filas y columnas para usarlas en la definición de la matriz:

```
#include <stdio.h>

#define FILAS      9
#define COLUMNAS   20

int main(void)
{
  char espacio[FILAS][COLUMNAS];

  return 0;
}
```

La matriz *espacio* es una matriz de caracteres. Hemos de inicializarla con caracteres '+', que indican que no se han explorado sus casillas. En lugar de inicializarla en *main*, vamos a diseñar una función especial para ello. ¿Por qué? Para mantener *main* razonablemente pequeño y mejorar así la legibilidad. A estas alturas no debe asustarnos definir funciones para las diferentes tareas.

```
#include <stdio.h>

#define FILAS      9
#define COLUMNAS   20

#define NO_SONDEADA '+'

void inicializa_tablero(char tablero[][COLUMNAS])
    /* Inicializa el tablero de juego marcando todas las casillas como no sondeadas. */
{
  int i, j;

  for (i=0; i<FILAS; i++)
    for (j=0; j<COLUMNAS; j++)
      tablero[i][j] = NO_SONDEADA;
}

int main(void)
{
  char espacio[FILAS][COLUMNAS];

  inicializa_tablero(espacio);

  return 0;
}
```

Pasamos la matriz indicando el número de columnas de la misma.[5] En el interior de la función se modifica el contenido de la matriz. Los cambios afectarán a la variable que suministremos como argumento, pues las matrices se pasan siempre por referencia.

Hemos de mostrar por pantalla el contenido de la matriz en más de una ocasión. Podemos diseñar un procedimiento que se encargue de esta tarea:

```
#include <stdio.h>

#define FILAS       9
#define COLUMNAS   20

#define NO_SONDEADA '+'

...

void muestra_tablero(char tablero[][COLUMNAS])
    /* Muestra en pantalla el tablero de juego. */
{
  int i, j;

  // Etiquetar con una letra cada columna.
  printf("␣␣");
  for (j=0; j<COLUMNAS; j++) printf("%c", 'A'+j);
  printf("\n");

  for (i=0; i<FILAS; i++) {
    printf("%d␣", i); // Etiqueta de cada fila.
    for (j=0; j<COLUMNAS; j++)
      printf("%c", tablero[i][j]);
    printf("\n");
  }
}

int main(void)
{
  char espacio[FILAS][COLUMNAS];

  inicializa_tablero(espacio);
  muestra_tablero(espacio);

  return 0;
}
```

El procedimiento *muestra_tablero* imprime, además, del contenido del tablero, el nombre de las columnas y el número de las filas.

Por cierto, hay una discrepancia entre el modo con que nos referimos a las casillas (mediante un dígito y una letra) y el modo con el que lo hace el programa (mediante dos números enteros). Cuando pidamos unas coordenadas al usuario lo haremos con una sentencia como ésta:

```
...
#define TALLACAD   80
...
int main(void)
{
```

[5] No hemos usado el nombre *espacio*, sino *tablero*, con el único objetivo de resaltar que el parámetro puede ser cualquier matriz (siempre que su dimensión se ajuste a lo esperado), aunque nosotros sólo usaremos la matriz *espacio* como argumento. Si hubiésemos usado el mismo nombre, es probable que hubiésemos alimentado la confusión entre parámetros y argumentos que experimentáis algunos.

```
  ...
  char coordenadas[TALLACAD+1];

  ...

  printf("Coordenadas:␣"); scanf("%s", coordenadas);
  ...
```

Como ves, las coordenadas se leerán en una cadena. Nos convendrá disponer, pues, de una función que «traduzca» esa cadena a un par de números y otra que haga lo contrario:

```
void de_fila_y_columna_a_numero_y_letra(int fila, int columna, char * coordenadas)
    /* Convierte una fila y columna descritas numéricamente en una fila y columna descritas
     * como una cadena con un dígito y una letra.
     */
{
  coordenadas[0] = '0' + fila;
  coordenadas[1] = 'A' + columna;
  coordenadas[2] = '\0';
}

int de_numero_y_letra_a_fila_y_columna(char coordenadas[], int * fila, int * columna)
    /* Convierte una fila y columna con un dígito y una letra (minúscula o mayúscula) en
     * cualquier orden a una fila y columna descritas numéricamente.
     */
{
  if (strlen(coordenadas) != 2)
    return 0;
  if (coordenadas[0] >= '0' && coordenadas[0] <= '8' && isalpha(coordenadas[1])) {
    *fila = coordenadas[0] - '0';
    *columna = toupper(coordenadas[1]) - 'A';
    return 1;
  }
  if (coordenadas[1] >= '0' && coordenadas[1] <= '8' && isalpha(coordenadas[0])) {
    *columna = toupper(coordenadas[0]) - 'A';
    *fila = coordenadas[1] - '0';
    return 1;
  }
  return 0;
}
```

La primera función (*de_fila_y_columna_a_numero_y_letra*) es muy sencilla: recibe el valor de la fila y el valor de la columna y modifica el contenido de un puntero a una cadena. Observa que es responsabilidad nuestra terminar correctamente la cadena *coordenadas*. La segunda función es algo más complicada. Una razón para ello es que efectúa cierto tratamiento de errores. ¿Por qué? Porque la cadena *coordenadas* ha sido introducida por el usuario y puede contener errores. Usamos un convenio muy frecuente en los programas C:

- Los valores se devuelven en la función mediante parámetros pasados por referencia,
- y la función devuelve un valor que indica si se detectó o no un error (devuelve 0 si hubo error, y 1 en caso contrario).

De este modo es posible invocar a la función cuando leemos el contenido de la cadena de esta forma:

```
  ...
  printf("Coordenadas:␣"); scanf("%s", coordenadas);
```

```
  while (!de_numero_y_letra_a_fila_y_columna(coordenadas, &fila, &columna)) {
    printf("Coordenadas␣no␣válidas.␣Inténtelo␣de␣nuevo.\nCoordenadas:␣");
    scanf("%s", coordenadas);
  }
  ...
```

Sigamos. Hemos de disponer ahora 5 náufragos en el tablero de juego. Podríamos ponerlos directamente en la matriz *espacio* modificando el valor de las casillas pertinentes, pero en tal caso *muestra_tablero* los mostraría, revelando el secreto de su posición y reduciendo notablemente el interés del juego ;-). ¿Qué hacer? Una posibilidad consiste en usar una matriz adicional en la que poder disponer los náufragos. Esta nueva matriz no se mostraría nunca al usuario y sería consultada por el programa cuando se necesitara saber si hay un náufrago en alguna posición determinada del tablero. Si bien es una posibilidad interesante (y te la propondremos más adelante como ejercicio), nos decantamos por seguir una diferente que nos permitirá practicar el paso de registros a funciones. Definiremos los siguientes registros:

```
...

#define MAX_NAUFRAGOS 5

struct Naufrago {
  int fila, columna; // Coordenadas
  int encontrado;    // ¿Ha sido encontrado ya?
};

struct GrupoNaufragos {
  struct Naufrago naufrago[MAX_NAUFRAGOS];
  int cantidad;
};

...
```

El tipo registro **struct** *Naufrago* mantiene la posición de un náufrago y permite saber si sigue perdido o si, por el contrario, ya ha sido encontrado. El tipo registro **struct** *GrupoNaufragos* mantiene un vector de náufragos de talla MAX_NAUFRAGOS. Aunque el juego indica que hemos de trabajar con 5 náufragos, usaremos un campo adicional con la cantidad de náufragos realmente almacenados en el vector. De ese modo resultará sencillo modificar el juego (como te proponemos en los ejercicios al final de esta sección) para que se juegue con un número de náufragos seleccionado por el usuario.

Guardaremos los náufragos en una variable de tipo **struct** *GrupoNaufragos*:

```
...

int main(void)
{
  char espacio[FILAS][COLUMNAS];
  struct GrupoNaufragos losNaufragos;

  inicializa_tablero(espacio);
  muestra_tablero(espacio);

  return 0;
}
```

El programa debería empezar realmente por inicializar el registro *losNaufragos* ubicando a cada náufrago en una posición aletoria del tablero. Esta función (errónea) se encarga de ello:

```
...
```

```
#include <stdlib.h>
...
void pon_naufragos(struct GrupoNaufragos * grupoNaufragos, int cantidad)
     /* Situa aleatoriamente cantidad náufragos en la estructura grupoNaufragos. */
     /* PERO LO HACE MAL. */
{
  int fila, columna;

  grupoNaufragos->cantidad = 0;
  while (grupoNaufragos->cantidad != cantidad) {
     fila = rand() % FILAS;
     columna = rand() % COLUMNAS;
     grupoNaufragos->naufrago[grupoNaufragos->cantidad].fila = fila;
     grupoNaufragos->naufrago[grupoNaufragos->cantidad].columna = columna;
     grupoNaufragos->naufrago[grupoNaufragos->cantidad].encontrado = 0;
     grupoNaufragos->cantidad++;
  }
}
```

¿Por qué está mal? Primero hemos de entenderla bien. Analicémosla paso a paso. Empecemos por la cabecera: la función tiene dos parámetros, uno que es una referencia (un puntero) a un registro de tipo **struct** *GrupoNaufragos* y un entero que nos indica cuántos náufragos hemos de poner al azar. La rutina empieza inicializando a cero la cantidad de náufragos ya dispuestos mediante una línea como ésta:

```
grupoNaufragos->cantidad = 0;
```

¿Entiendes por qué se usa el operador flecha?: la variable *grupoNaufragos* es un puntero, así que hemos de acceder a la información apuntada antes de acceder al campo *cantidad*. Podríamos haber escrito esa misma línea así:

```
(*grupoNaufragos).cantidad = 0;
```

pero hubiera resultado más incómodo (e ilegible). A continuación, la función repite *cantidad* veces la acción consistente en seleccionar una fila y columna al azar (mediante la función *rand* de `stdlib.h`) y lo anota en una posición del vector de náufragos. Puede que esta línea te resulte un tanto difícil de entender:

```
grupoNaufragos->naufrago[grupoNaufragos->cantidad].fila = fila;
```

pero no lo es tanto si la analizas paso a paso. Veamos. Empecemos por el índice que hemos sombreado arriba. La primera vez, es 0, la segunda 1, y así sucesivamente. En aras de comprender la sentencia, nos conviene reescribir la sentencia poniendo de momento un 0 en el índice:

```
grupoNaufragos->naufrago[0].fila = fila;
```

Más claro, ¿no? Piensa que *grupoNaufragos*->*naufrago* es un vector como cualquier otro, así que la expresión *grupoNaufragos*->*naufrago*[0] accede a su primer elemento. ¿De qué tipo es ese elemento? De tipo **struct** *Naufrago*. Un elemento de ese tipo tiene un campo *fila* y se accede a él con el operador punto. O sea, esa sentencia asigna el valor de *fila* al campo *fila* de un elemento del vector *naufrago* del registro que es apuntado por *grupoNaufragos*. El resto de la función te debe resultar fácil de leer ahora. Volvamos a la cuestión principal: ¿por qué está mal diseñada esa función? Fácil: porque puede ubicar dos náufragos en la misma casilla del tablero. ¿Cómo corregimos el problema? Asegurándonos de que cada náufrago ocupa una casilla diferente. Tenemos dos posibilidades:

- Generar las posiciones de cinco náufragos al azar y comprobar que son todas diferentes entre sí. Si lo son, perfecto: hemos acabado; si no, volvemos a repetir todo el proceso.

- Ir generando la posición de cada náufrago de una en una, comprobando cada vez que ésta es distinta de la de todos los náufragos anteriores. Si no lo es, volvemos a generar la posición de este náufrago concreto; si lo es, pasamos al siguiente.

La segunda resulta más sencilla de implementar y es, a la vez, más eficiente. Aquí la tienes implementada:

```
void pon_naufragos(struct GrupoNaufragos * grupoNaufragos, int cantidad)
    /* Sitúa aleatoriamente cantidad náufragos en la estructura grupoNaufragos. */
{
  int fila, columna, ya_hay_uno_ahi, i;

  grupoNaufragos->cantidad = 0;
  while (grupoNaufragos->cantidad != cantidad) {
    fila = rand() % FILAS;
    columna = rand() % COLUMNAS;
    ya_hay_uno_ahi = 0;
    for (i=0; i<grupoNaufragos->cantidad; i++)
      if (fila == grupoNaufragos->naufrago[i].fila &&
          columna == grupoNaufragos->naufrago[i].columna) {
        ya_hay_uno_ahi = 1;
        break;
      }
    if (!ya_hay_uno_ahi) {
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].fila = fila;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].columna = columna;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].encontrado = 0;
      grupoNaufragos->cantidad++;
    }
  }
}
```

Nos vendrá bien disponer de una función que muestre por pantalla la ubicación y estado de cada náufrago. Esta función no resulta útil para el juego (pues perdería toda la gracia), pero sí para ayudarnos a depurar el programa. Podríamos, por ejemplo, ayudarnos con llamadas a esa función mientras jugamos partidas de prueba y, una vez dado por bueno el programa, no llamarla más. En cualquier caso, aquí la tienes:

```
void muestra_naufragos(struct GrupoNaufragos grupoNaufragos)
    /* Muestra las coordenadas de cada náufrago e informa de si sigue perdido.
     * Útil para depuración del programa.
     */
{
  int i;
  char coordenadas[3];

  for (i=0; i<grupoNaufragos.cantidad; i++) {
    de_fila_y_columna_a_numero_y_letra(grupoNaufragos.naufrago[i].fila,
  grupoNaufragos.naufrago[i].columna,
  coordenadas);
    printf("Náufrago %d en coordenadas %s ", i, coordenadas);
    if (grupoNaufragos.naufrago[i].encontrado)
      printf("ya ha sido encontrado.\n");
    else
      printf("sigue perdido.\n");
  }
}
```

La función está bien, pero podemos mejorarla. Fíjate en cómo pasamos su parámetro: por valor. ¿Por qué? Porque no vamos a modificar su valor en el interior de la función.

En principio, la decisión de pasarlo por valor está bien fundamentada. No obstante, piensa en qué ocurre cada vez que llamamos a la función: como un registro de tipo **struct** *GrupoNaufragos* ocupa 64 bytes (haz cuentas y compruébalo), cada llamada a la función obliga a copiar 64 bytes en la pila. El problema se agravaría si en lugar de trabajar con un número máximo de 5 náufragos lo hiciéramos con una cantidad mayor. ¿Es realmente necesario ese esfuerzo? La verdad es que no: podemos limitarnos a copiar 4 bytes si pasamos una referencia al registro. Esta nueva versión de la función efectúa el paso por referencia:

```
void muestra_naufragos(struct GrupoNaufragos * grupoNaufragos)
    /* Muestra las coordenadas de cada náufrago e informa de si sigue perdido.
     * Útil para depuración del programa.
     */
{
  int i, fila, columna;
  char coordenadas[3];

  for (i=0; i<grupoNaufragos->cantidad; i++) {
    de_fila_y_columna_a_numero_y_letra(grupoNaufragos->naufrago[i].fila,
  grupoNaufragos->naufrago[i].columna,
  coordenadas);
    printf ("Náufrago %d en coordenadas %s ", i, coordenadas);
    if (grupoNaufragos->naufrago[i].encontrado)
      printf ("ya ha sido encontrado.\n");
    else
      printf ("sigue perdido.\n");
  }
}
```

Es posible usar el adjetivo **const** para dejar claro que pasamos el puntero por eficiencia, pero no porque vayamos a modificar su contenido:

```
void muestra_naufragos(const struct GrupoNaufragos * grupoNaufragos)
```

Hagamos una prueba para ver si todo va bien por el momento:

```
...

int main(void)
{
  struct GrupoNaufragos losNaufragos;

  pon_naufragos(&losNaufragos, 5);
  muestra_naufragos(&losNaufragos);

  return 0;
}
```

Compilemos y ejecutemos el programa. He aquí el resultado:

```
$ gcc minigalaxis.c -o minigalaxis ↲
$ minigalaxis ↲
Náufrago 0 en coordenadas 1G sigue perdido.
Náufrago 1 en coordenadas 0P sigue perdido.
Náufrago 2 en coordenadas 5P sigue perdido.
Náufrago 3 en coordenadas 1M sigue perdido.
Náufrago 4 en coordenadas 6B sigue perdido.
```

Bien: cada náufrago ocupa una posición diferente. Ejecutémoslo de nuevo

```
$ minigalaxis↲
Náufrago 0 en coordenadas 1G sigue perdido.
Náufrago 1 en coordenadas 0P sigue perdido.
Náufrago 2 en coordenadas 5P sigue perdido.
Náufrago 3 en coordenadas 1M sigue perdido.
Náufrago 4 en coordenadas 6B sigue perdido.
```

¡Eh! ¡Se han ubicado en las mismas posiciones! ¿Qué gracia tiene el juego si en todas las partidas aparecen los náufragos en las mismas casillas? ¿Cómo es posible que ocurra algo así? ¿No se generaba su ubicación al azar? Sí y no. La función *rand* genera números *pseudo*aleatorios. Utiliza una fórmula matemática que genera una secuencia de números de forma tal que no podemos efectuar una predicción del siguiente (a menos que conozcamos la fórmula, claro está). La secuencia de números se genera a partir de un número inicial: la *semilla*. En principio, la semilla es siempre la misma, así que la secuencia de números es, también, siempre la misma. ¿Qué hacer, pues, si queremos obtener una diferente? Una posibilidad es solicitar al usuario el valor de la semilla, que se puede modificar con la función *srand*, pero no parece lo adecuado para un juego de ordenador (el usuario podría hacer trampa introduciendo siempre la misma semilla). Otra posibilidad es inicializar la semilla con un valor aleatorio. ¿Con un valor aleatorio? Tenemos un pez que se muerde la cola: ¡resulta que necesito un número aleatorio para generar números aleatorios! Mmmmm. Tranquilo, hay una solución: consultar el reloj del ordenador y usar su valor como semilla. La función *time* (disponible incluyendo `time.h`) nos devuelve el número de segundos transcurridos desde el inicio del día 1 de enero de 1970 (lo que se conoce por tiempo de la era Unix) y, naturalmente, es diferente cada vez que lo llamamos para iniciar una partida. Aquí tienes la solución:

```c
...
#include <time.h>
...

int main(void)
{
  struct GrupoNaufragos losNaufragos;

  srand(time(0));

  pon_naufragos(&losNaufragos, 5);
  muestra_naufragos(&losNaufragos);

  return 0;
}
```

Efectuemos nuevas pruebas:

```
$ gcc minigalaxis.c -o minigalaxis↲
$ minigalaxis↲
Náufrago 0 en coordenadas 6K sigue perdido.
Náufrago 1 en coordenadas 5L sigue perdido.
Náufrago 2 en coordenadas 6E sigue perdido.
Náufrago 3 en coordenadas 3I sigue perdido.
Náufrago 4 en coordenadas 8T sigue perdido.
```

¡Bravo! Son valores diferentes de los anteriores. Ejecutemos nuevamente el programa:

```
$ minigalaxis↲
Náufrago 0 en coordenadas 2D sigue perdido.
Náufrago 1 en coordenadas 4H sigue perdido.
Náufrago 2 en coordenadas 5J sigue perdido.
Náufrago 3 en coordenadas 4E sigue perdido.
Náufrago 4 en coordenadas 7G sigue perdido.
```

¡Perfecto! A otra cosa.

Ya hemos inicializado el tablero y dispuesto los náufragos en posiciones al azar. Diseñemos una función para el lanzamiento de sondas. La función (que será un procedimiento) recibirá un par de coordenadas, el tablero de juego y el registro que contiene la posición de los náufragos y hará lo siguiente:

- modificará el tablero de juego sustituyendo los símbolos `'+'` por `'.'` en las direcciones cardinales desde el punto de lanzamiento de la sonda,
- y modificará la casilla en la que se lanzó la sonda indicando el número de náufragos detectados, o marcándola con una `'X'` si hay un náufrago en ella.

```
...
#define NO_SONDEADA '+'
#define RESCATADO   'X'
#define SONDEADA    '.'
...

void lanzar_sonda(int fila, int columna, char tablero[][COLUMNAS],
                  const struct GrupoNaufragos * grupoNaufragos)
    /* Lanza sonda en las coordenadas indicadas. Actualiza el tablero con el resultado del
     * sondeo. Si se detecta un náufrago en el punto de lanzamiento de la sonda, lo rescata.
     */
{
  int detectados = 0, i;

  // Recorrer la vertical
  for (i=0; i<FILAS; i++) {
    if (hay_naufrago(i, columna, grupoNaufragos))
      detectados++;
    if (tablero[i][columna] == NO_SONDEADA)
      tablero[i][columna] = SONDEADA;
  }

  // Recorrer la horizontal
  for (i=0; i<COLUMNAS; i++) {
    if (hay_naufrago(fila, i, grupoNaufragos))
      detectados++;
    if (tablero[fila][i] == NO_SONDEADA)
      tablero[fila][i] = SONDEADA;
  }

  // Ver si acertamos y hay un náufrago en esta misma casilla.
  if (hay_naufrago(fila, columna, grupoNaufragos)) {
    tablero[fila][columna] = RESCATADO; // En tal caso, ponemos una X.
    rescate(fila, columna, grupoNaufragos);
  }
  else
    tablero[fila][columna] = '0' + detectados; // Y si no, el número de náufragos detectados.
}
```

Esta función se ayuda con otras dos: *hay_naufrago* y *rescate*. La primera nos indica si hay un náufrago en una casilla determinada:

```
int hay_naufrago(int fila, int columna, const struct GrupoNaufragos * grupoNaufragos)
    /* Averigua si hay un náufrago perdido en las coordenadas (fila, columna).
     * Si lo hay devuelve 1; si no lo hay, devuelve 0.
     */
```

```
{
   int i;

   for (i=0; i<grupoNaufragos->cantidad; i++)
      if (fila == grupoNaufragos->naufrago[i].fila &&
            columna == grupoNaufragos->naufrago[i].columna)
         return 1;
   return 0;
}
```

Y la segunda lo marca como rescatado:

```
void rescate(int fila, int columna, struct GrupoNaufragos * grupoNaufragos)
      /* Rescata al náufrago que hay en las coordenadas indicadas. */
{
   int i;

   for (i=0; i<grupoNaufragos->cantidad; i++)
      if (fila == grupoNaufragos->naufrago[i].fila &&
            columna == grupoNaufragos->naufrago[i].columna)
         grupoNaufragos->naufrago[i].encontrado = 1;
}
```

Ya podemos ofrecer una versión más completa del programa principal:

```
int main(void)
{
   char espacio[FILAS][COLUMNAS];
   struct GrupoNaufragos losNaufragos;
   char coordenadas[TALLACAD+1];
   int fila, columna;

   srand(time(0));

   pon_naufragos(&losNaufragos, 5);
   inicializa_tablero(espacio);
   muestra_tablero(espacio);

   while ( ??? ) {
      printf("Coordenadas:␣"); scanf("%s", coordenadas);
      while (!de_numero_y_letra_a_fila_y_columna(coordenadas, &fila, &columna)) {
         printf("Coordenadas␣no␣válidas.␣Inténtelo␣de␣nuevo.\nCoordenadas:␣");
         scanf("%s", coordenadas);
      }
      lanzar_sonda(fila, columna, espacio, &losNaufragos);
      muestra_tablero(espacio);
   }

   return 0;
}
```

¿Cuándo debe finalizar el bucle **while** exterior? Bien cuando hayamos rescatado a todos los náufragos, bien cuando nos hayamos quedado sin sondas. En el primer caso habremos vencido y en el segundo habremos perdido:

```
...
#define SONDAS 20
...

int perdidos(const struct GrupoNaufragos * grupoNaufragos)
```

```
    /* Cuenta el número de náufragos que siguen perdidos. */
{
  int contador = 0, i;

  for (i=0; i<grupoNaufragos->cantidad; i++)
    if (!grupoNaufragos->naufrago[i].encontrado)
      contador++;
  return contador;
}

...

int main(void)
{
  char espacio[FILAS][COLUMNAS];
  struct GrupoNaufragos losNaufragos;
  int sondas_disponibles = SONDAS;
  char coordenadas[TALLACAD+1];
  int fila, columna;

  srand(time(0));

  pon_naufragos(&losNaufragos, 5);
  inicializa_tablero(espacio);
  muestra_tablero(espacio);

  while (sondas_disponibles > 0 && perdidos(&losNaufragos) > 0) {
    printf("Hay␣%d␣náufragos\n", perdidos(&losNaufragos));
    printf("Dispones␣de␣%d␣sondas\n", sondas_disponibles);
    printf("Coordenadas:␣"); scanf("%s", coordenadas);
    while (!de_numero_y_letra_a_fila_y_columna(coordenadas, &fila, &columna)) {
      printf("Coordenadas␣no␣válidas.␣Inténtelo␣de␣nuevo.\nCoordenadas:␣");
      scanf("%s", coordenadas);
    }
    lanzar_sonda(fila, columna, espacio, &losNaufragos);
    muestra_tablero(espacio);
    sondas_disponibles--;
  }

  if (perdidos(&losNaufragos) == 0)
    printf("Has␣ganado.␣Puntuación:␣%d␣puntos.\n", SONDAS - sondas_disponibles);
  else
    printf("Has␣perdido.␣Por␣tu␣culpa␣han␣muerto␣%d␣náufragos\n",
           perdidos(&losNaufragos));

  return 0;
}
```

Hemos definido una nueva función, *perdidos*, que calcula el número de náufragos que permanecen perdidos.

Y ya está. Te mostramos finalmente el listado completo del programa:

minigalaxis.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

```

```c
#define FILAS        9
#define COLUMNAS     20
#define TALLACAD     80
#define MAX_NAUFRAGOS 5
#define SONDAS       20

#define NO_SONDEADA '+'
#define RESCATADO   'X'
#define SONDEADA    '.'

/**********************************************************
 * Conversión entre los dos modos de expresar coordenadas
 **********************************************************/

void de_fila_y_columna_a_numero_y_letra(int fila, int columna, char coordenadas[])
    /* Convierte fila y columna descritas numéricamente en fila y columna descritas
     * como una cadena con un dígito y una letra.
     */
{
  coordenadas[0] = '0' + fila;
  coordenadas[1] = 'A' + columna;
  coordenadas[2] = '\0';
}

int de_numero_y_letra_a_fila_y_columna(char coordenadas[], int * fila, int * columna)
    /* Convierte una fila y columna con un dígito y una letra (minúscula o mayúscula) en
     * cualquier orden a una fila y columna descritas numéricamente.
     */
{
  printf(">>> %s\n", coordenadas);
  if (strlen(coordenadas) != 2)
    return 0;
  if (coordenadas[0] >= '0' && coordenadas[0] <= '8' && isalpha(coordenadas[1])) {
    *fila = coordenadas[0] - '0';
    *columna = toupper(coordenadas[1]) - 'A';
    return 1;
  }
  if (coordenadas[1] >= '0' && coordenadas[1] <= '8' && isalpha(coordenadas[0])) {
    *columna = toupper(coordenadas[0]) - 'A';
    *fila = coordenadas[1] - '0';
    return 1;
  }
  return 0;
}

/****************************************
 * Náufragos
 ****************************************/

struct Naufrago {
  int fila, columna; // Coordenadas
  int encontrado;    // ¿Ha sido encontrado ya?
};

struct GrupoNaufragos {
  struct Naufrago naufrago[MAX_NAUFRAGOS];
  int cantidad;
};

```

```
void pon_naufragos(struct GrupoNaufragos * grupoNaufragos, int cantidad)
     /* Situa aleatoriamente cantidad náufragos en la estructura grupoNaufragos. */
{
  int fila, columna, ya_hay_uno_ahi, i;

  grupoNaufragos->cantidad = 0;
  while (grupoNaufragos->cantidad != cantidad) {
    fila = rand() % FILAS;
    columna = rand() % COLUMNAS;
    ya_hay_uno_ahi = 0;
    for (i=0; i<grupoNaufragos->cantidad; i++)
      if (fila == grupoNaufragos->naufrago[i].fila &&
            columna == grupoNaufragos->naufrago[i].columna) {
        ya_hay_uno_ahi = 1;
        break;
      }
    if (!ya_hay_uno_ahi) {
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].fila = fila;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].columna = columna;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].encontrado = 0;
      grupoNaufragos->cantidad++;
    }
  }
}

int hay_naufrago(int fila, int columna, const struct GrupoNaufragos * grupoNaufragos)
     /* Averigua si hay un náufrago perdido en las coordenadas (fila, columna).
      * Si lo hay devuelve 1; si no lo hay, devuelve 0.
      */
{
  int i;

  for (i=0; i<grupoNaufragos->cantidad; i++)
    if (fila == grupoNaufragos->naufrago[i].fila &&
          columna == grupoNaufragos->naufrago[i].columna)
      return 1;
  return 0;
}


void rescate(int fila, int columna, struct GrupoNaufragos * grupoNaufragos)
     /* Rescata al náufrago que hay en las coordenadas indicadas. */
{
  int i;

  for (i=0; i<grupoNaufragos->cantidad; i++)
    if (fila == grupoNaufragos->naufrago[i].fila &&
          columna == grupoNaufragos->naufrago[i].columna)
      grupoNaufragos->naufrago[i].encontrado = 1;
}

int perdidos(const struct GrupoNaufragos * grupoNaufragos)
     /* Cuenta el número de náufragos que siguen perdidos. */
{
  int contador = 0, i;

  for (i=0; i<grupoNaufragos->cantidad; i++)
    if (!grupoNaufragos->naufrago[i].encontrado)
      contador++;
```

```
  return contador;
}

void muestra_naufragos(const struct GrupoNaufragos * grupoNaufragos)
    /* Muestra las coordenadas de cada naufrago e informa de si sigue perdido.
     * Útil para depuración del programa.
     */
{
  int i;
  char coordenadas[3];

  for (i=0; i<grupoNaufragos->cantidad; i++) {
    de_fila_y_columna_a_numero_y_letra(grupoNaufragos->naufrago[i].fila,
  grupoNaufragos->naufrago[i].columna,
  coordenadas);
    printf("Naufrago %d en coordenadas %s ", i, coordenadas);
    if (grupoNaufragos->naufrago[i].encontrado)
      printf("ya ha sido encontrado.\n");
    else
      printf("sigue perdido.\n");
  }
}

/****************************************
 * Tablero
 ****************************************/

void inicializa_tablero(char tablero[][COLUMNAS])
    /* Inicializa el tablero de juego marcando todas las casillas como no sondeadas. */
{
  int i, j;

  for (i=0; i<FILAS; i++)
    for (j=0; j<COLUMNAS; j++)
      tablero[i][j] = NO_SONDEADA;
}

void muestra_tablero(char tablero[][COLUMNAS])
    /* Muestra en pantalla el tablero de juego. */
{
  int i, j;

  // Etiquetar con una letra cada columna.
  printf("  ");
  for (j=0; j<COLUMNAS; j++) printf("%c", 'A'+j);
  printf("\n");

  for (i=0; i<FILAS; i++) {
    printf("%d ", i); // Etiqueta de cada fila.
    for (j=0; j<COLUMNAS; j++)
      printf("%c", tablero[i][j]);
    printf("\n");
  }
}

/****************************************
 * Sonda
 ****************************************/
```

```
void lanzar_sonda(int fila, int columna, char tablero[][COLUMNAS],
                  struct GrupoNaufragos * grupoNaufragos)
    /* Lanza sonda en las coordenadas indicadas. Actualiza el tablero con el resultado del
     * sondeo. Si se detecta un náufrago en el punto de lanzamiento de la sonda, lo rescata.
     */
{
  int detectados = 0, i;

  // Recorrer la vertical
  for (i=0; i<FILAS; i++) {
    if (hay_naufrago(i, columna, grupoNaufragos))
      detectados++;
    if (tablero[i][columna] == NO_SONDEADA)
      tablero[i][columna] = SONDEADA;
  }

  // Recorrer la horizontal
  for (i=0; i<COLUMNAS; i++) {
    if (hay_naufrago(fila, i, grupoNaufragos))
      detectados++;
    if (tablero[fila][i] == NO_SONDEADA)
      tablero[fila][i] = SONDEADA;
  }

  // Ver si acertamos y hay una náufrago en esta misma casilla.
  if (hay_naufrago(fila, columna, grupoNaufragos)) {
    tablero[fila][columna] = RESCATADO; // En tal caso, ponemos una X.
    rescate(fila, columna, grupoNaufragos);
  }
  else
    tablero[fila][columna] = '0' + detectados; // Si no, número de náufragos detectados.
}

int main(void)
{
  char espacio[FILAS][COLUMNAS];
  struct GrupoNaufragos losNaufragos;
  int sondas_disponibles = SONDAS;
  char coordenadas[TALLACAD+1];
  int fila, columna;

  srand(time(0));

  pon_naufragos(&losNaufragos, 5);
  inicializa_tablero(espacio);
  muestra_tablero(espacio);

  while (sondas_disponibles > 0 && perdidos(&losNaufragos) > 0) {
    printf("Hay␣%d␣náufragos\n", perdidos(&losNaufragos));
    printf("Dispones␣de␣%d␣sondas\n", sondas_disponibles);
    printf("Coordenadas:␣"); scanf("%s", coordenadas);
    while (!de_numero_y_letra_a_fila_y_columna(coordenadas, &fila, &columna)) {
      printf("Coordenadas␣no␣válidas.␣Inténtelo␣de␣nuevo.\nCoordenadas:␣");
      scanf("%s", coordenadas);
    }
    lanzar_sonda(fila, columna, espacio, &losNaufragos);
    muestra_tablero(espacio);
    sondas_disponibles--;
  }
```

```

  if (perdidos(&losNaufragos) == 0)
    printf("Has␣ganado.␣Puntuación:␣%d␣puntos.\n", SONDAS - sondas_disponibles);
  else
    printf("Has␣perdido.␣Por␣tu␣culpa␣han␣muerto␣%d␣náufragos\n",
           perdidos(&losNaufragos));

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 202 Reescribe el programa para que no se use una variable de tipo **struct** *GrupoNaufragos* como almacén del grupo de náufragos, sino una matriz paralela a la matriz *espacio*.

Cada náufrago se representará con un `'*'` mientras permanezca perdido, y con una `'X'` cuando haya sido descubierto.

▶ 203 Siempre que usamos *rand* en miniGalaxis calculamos un par de números aleatorios. Hemos definido un nuevo tipo y una función:

```
struct Casilla {
  int fila, columna;
};

struct Casilla casilla_al_azar(void)
{
  struct Casilla casilla;

  casilla.fila = rand() % FILAS;
  casilla.columna = rand() % COLUMNAS;
  return casilla;
}
```

Y proponemos usarlos así:

```
void pon_naufragos(struct GrupoNaufragos * grupoNaufragos, int cantidad)
    /* Situa aleatoriamente cantidad náufragos en la estructura grupoNaufragos. */
{
  int fila, columna, ya_hay_uno_ahi, i;
  struct Casilla casilla;

  grupoNaufragos->cantidad = 0;
  while (grupoNaufragos->cantidad != cantidad) {
    casilla = casilla_al_azar();
    ya_hay_uno_ahi = 0;
    for (i=0; i<grupoNaufragos->cantidad; i++)
      if (casilla.fila == grupoNaufragos->naufrago[i].fila &&
          casilla.columna == grupoNaufragos->naufrago[i].columna) {
        ya_hay_uno_ahi = 1;
        break;
      }
    if (!ya_hay_uno_ahi) {
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].fila = casilla.fila;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].columna = casilla.columna;
      grupoNaufragos->naufrago[grupoNaufragos->cantidad].encontrado = 0;
      grupoNaufragos->cantidad++;
    }
  }
}
```

¿Es correcto el programa con estos cambios?

▶ 204 Como siempre que usamos *rand* calculamos un par de números aleatorios, hemos modificado el programa de este modo:

```
struct Naufrago naufrago_al_azar(void)
{
  struct Naufrago naufrago;

  naufrago.fila = rand() % FILAS;
  naufrago.columna = rand() % COLUMNAS;
  naufrago.encontrado = 0;
  return naufrago;
}

void pon_naufragos(struct GrupoNaufragos * grupoNaufragos, int cantidad)
    /* Situa aleatoriamente cantidad náufragos en la estructura grupoNaufragos. */
{
  int fila, columna, ya_hay_uno_ahi, i;
  struct Naufrago un_naufrago;

  grupoNaufragos->cantidad = 0;
  while (grupoNaufragos->cantidad != cantidad) {
    un_naufrago = naufrago_al_azar();
    ya_hay_uno_ahi = 0;
    for (i=0; i<grupoNaufragos->cantidad; i++)
      if (un_naufrago.fila == grupoNaufragos->naufrago[i].fila &&
          un_naufrago.columna == grupoNaufragos->naufrago[i].columna) {
        ya_hay_uno_ahi = 1;
        break;
      }
    if (!ya_hay_uno_ahi) {
      grupoNaufragos->naufrago[grupoNaufragos->cantidad] = un_naufrago;
      grupoNaufragos->cantidad++;
    }
  }
}
```

¿Es correcto el programa con estos cambios?

▶ 205 Modifica el juego para que el usuario pueda escoger el nivel de dificultad. El usuario escogerá el número de náufragos perdidos (con un máximo de 20) y el número de sondas disponibles.

▶ 206 Hemos construido una versión simplificada de Galaxis. El juego original sólo se diferencia de éste en las direcciones exploradas por la sonda: así como las sondas de miniGalaxis exploran 4 direcciones, las de Galaxis exploran 8. Te mostramos el resultado de lanzar nuestra primera sonda en las coordenadas 4J de un tablero de juego Galaxis:

```
  ABCDEFGHIJKLMNOPQRST
0 +++++.+++.+++.++++++
1 ++++++.++.++.+++++++
2 +++++++.+.+.++++++++
3 ++++++++...+++++++++
4 .........1..........
5 ++++++++...+++++++++
6 +++++++.+.+.++++++++
7 ++++++.++.++.+++++++
8 +++++.+++.+++.++++++
```

Implementa el juego Galaxis.

. . . .

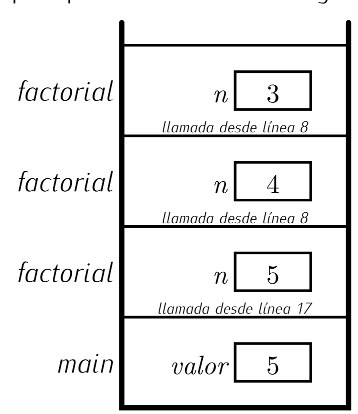
3.6. Recursión

Es posible definir funciones recursivas en C. La función *factorial* de este programa, por ejemplo, define un cálculo recursivo del factorial:

factorial_recursivo.c factorial_recursivo.c

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n<=1)
6      return 1;
7    else
8      return n * factorial (n-1);
9  }
10
11 int main(void)
12 {
13   int valor;
14
15   printf ("Dame␣un␣número␣entero␣positivo:␣");
16   scanf ("%d", &valor);
17   printf ("El␣factorial␣de␣%d␣vale:␣%d\n", valor, factorial(valor));
18
19   return 0;
20 }
```

Nada nuevo. Ya conoces el concepto de recursión de Python. En C es lo mismo. Tiene interés, eso sí, que estudiemos brevemente el aspecto de la memoria en un instante dado. Por ejemplo, cuando llamamos a *factorial*(5), que ha llamado a *factorial*(4), que a su vez ha llamado a *factorial*(3), la pila presentará esta configuración:



. . . . EJERCICIOS . . . .

▶ 207 Diseña una función que calcule recursivamente x^n. La variable x será de tipo **float** y n de tipo **int**.

▶ 208 Diseña una función recursiva que calcule el n-ésimo número de Fibonacci.

▶ 209 Diseña una función recursiva para calcular el número combinatorio n sobre m sabiendo que

$$\begin{aligned}\binom{n}{n} &= 1,\\ \binom{n}{0} &= 1,\\ \binom{n}{m} &= \binom{n-1}{m} + \binom{n-1}{m-1}.\end{aligned}$$

▶ 210 Diseña un procedimiento recursivo llamado *muestra_bin* que reciba un número entero positivo y muestre por pantalla su codificación en binario. Por ejemplo, si llamamos a *muestra_bin*(5), por pantalla aparecerá el texto «101».

3.6.1. Un método recursivo de ordenación: mergesort

Vamos a estudiar ahora un método recursivo de ordenación de vectores: *mergesort* (que se podría traducir por *ordenación por fusión o mezcla*). Estudiemos primero la aproximación que sigue considerando un procedimiento equivalente para ordenar las 12 cartas de un palo de la baraja de cartas. La ordenación por fusión de un palo de la baraja consiste en lo siguiente:

- Dividir el paquete de cartas en dos grupos de 6 cartas;
- *ordenar por fusión* el primer grupo de 6 cartas;
- *ordenar por fusión* el segundo grupo de 6 cartas;
- fundir los dos grupos, que ya están ordenados, tomando siempre la carta con número menor de cualquiera de los dos grupos (que siempre será la primera de uno de los dos grupos).

Ya ves dónde aparece la recursión, ¿no? Para ordenar 12 cartas por fusión hemos de ordenar dos grupos de 6 cartas por fusión. Y para ordenar cada grupo de 6 cartas por fusión tendremos que ordenar dos grupos de 3 cartas por fusión. Y para ordenar 3 grupos de cartas por fusión... ¿Cuándo finaliza la recursión? Cuando nos enfrentemos a casos triviales. Ordenar un grupo de 1 sola carta es trivial: ¡siempre está ordenado!

Desarrollemos un ejemplo de ordenación de un vector con 16 elementos:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
11	21	3	1	98	0	12	82	29	30	11	18	43	4	75	37

1. Empezamos separando el vector en dos «subvectores» de 8 elementos:

0	1	2	3	4	5	6	7
11	21	3	1	98	0	12	82

8	9	10	11	12	13	14	15
29	30	11	18	43	4	75	37

2. ordenamos por fusión el primer vector, con lo que obtenemos:

0	1	2	3	4	5	6	7
0	1	3	11	12	21	82	98

3. y ordenamos por fusión el segundo vector, con lo que obtenemos:

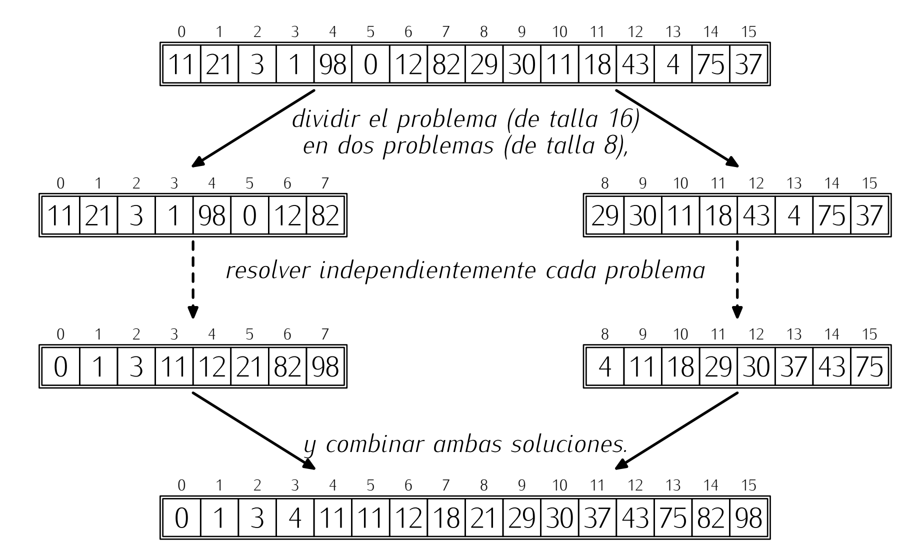
0	1	2	3	4	5	6	7
4	11	18	29	30	37	43	75

4. y ahora «fundimos» ambos vectores ordenados, obteniendo así un único vector ordenado:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	3	4	11	11	12	18	21	29	30	37	43	75	82	98

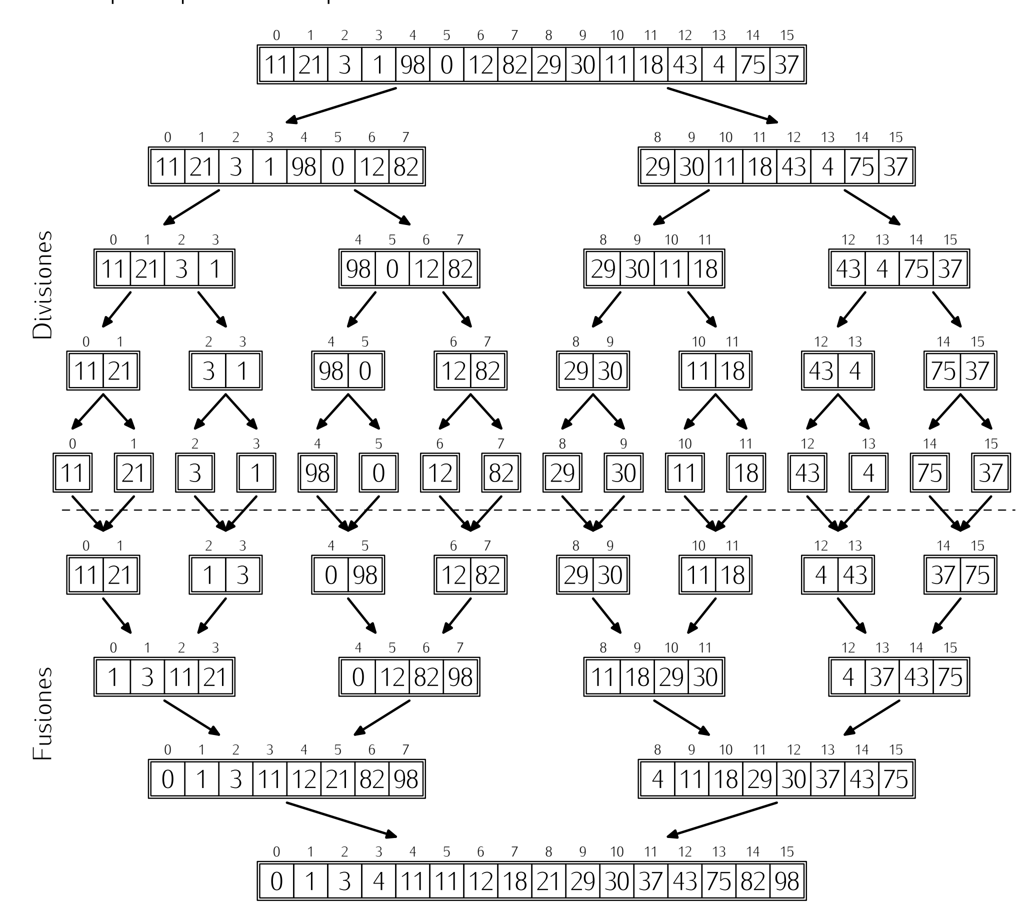
La idea básica de la fusión es sencilla: se recorren ambos vectores de izquierda a derecha, seleccionando en cada momento el menor elemento posible. Los detalles del proceso de fusión son un tanto escabrosos, así que lo estudiaremos con calma un poco más adelante.

Podemos representar el proceso realizado con esta imagen gráfica:



Está claro que hemos hecho «trampa»: las líneas de trazo discontinuo esconden un proceso complejo, pues la ordenación de cada uno de los vectores de 8 elementos supone la ordenación (recursiva) de dos vectores de 4 elementos, que a su vez… ¿Cuándo acaba el proceso recursivo? Cuando llegamos a un caso trivial: la ordenación de un vector que sólo tenga 1 elemento.

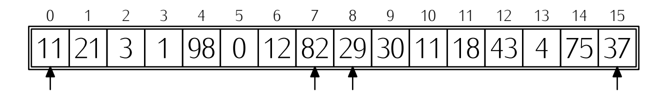
He aquí el proceso completo:



Nos queda por estudiar con detalle el proceso de fusión. Desarrollemos primero una función que recoja la idea básica de la ordenación por fusión: se llamará *mergesort* y recibirá un vector *v* y, en principio, la talla del vector que deseamos ordenar. Esta función utilizará una función auxiliar *merge* encargada de efectuar la fusión de vectores ya ordenados. Aquí tienes un borrador incompleto:

```
void mergesort(int v[], int talla)
{
  if (talla == 1)
    return;
  else {
    mergesort ( la primera mitad de v );
    mergesort ( la segunda mitad de v );
    merge(la primera mitad de v, la segunda mitad de v);
  }
}
```

Dejemos para más adelante el desarrollo de *merge*. De momento, el principal problema es cómo expresar lo de «la primera mitad de *v*» y «la segunda mitad de *v*». Fíjate: en el fondo, se trata de señalar una serie de elementos consecutivos del vector *v*. Cuando ordenábamos el vector del ejemplo teníamos:

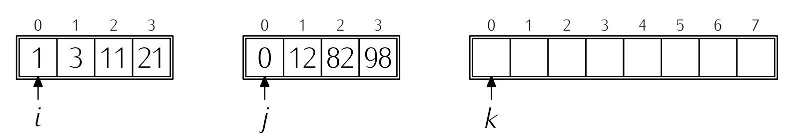


El primer «subvector» es la serie de valores entre el primer par de flechas, y el segundo «subvector» es la serie entre el segundo par de flechas. Modifiquemos, pues, *mergesort* para que trabaje con «subvectores», es decir, con un vector e índices que señalan dónde empieza y dónde acaba cada serie de valores.

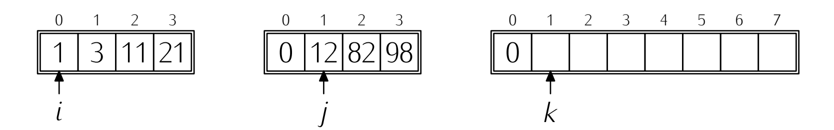
```
void mergesort(int v[], int inicio, int final)
{
  if (final - inicio == 0)
    return;
  else {
    mergesort ( v, inicio, (inicio+final) / 2);
    mergesort ( v, (inicio+final) / 2 + 1, final);
    merge(la primera mitad de v, la segunda mitad de v);
  }
}
```

Perfecto. Acabamos de expresar la idea de dividir un vector en dos sin necesidad de utilizar nuevos vectores.

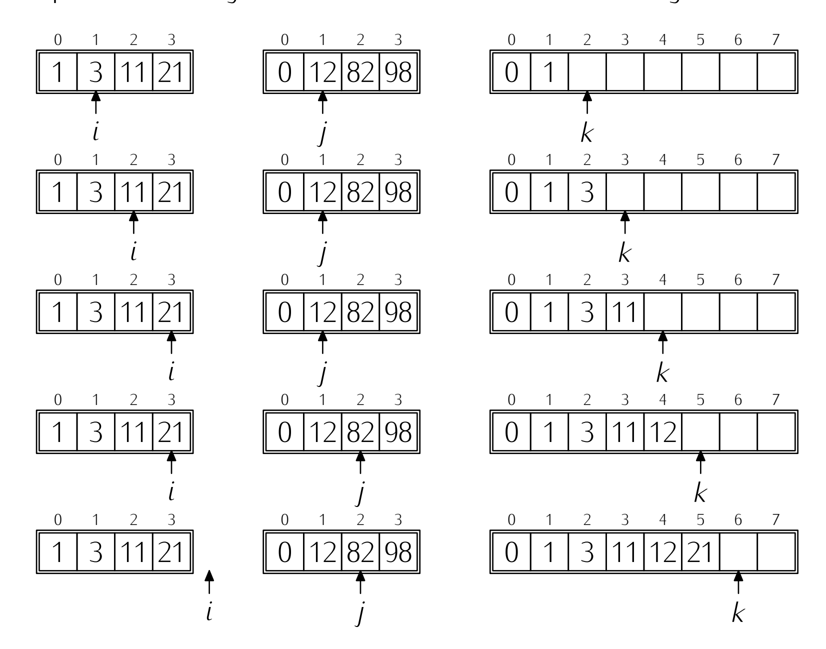
Nos queda por detallar la función *merge*. Dicha función recibe dos «subvectores» contiguos ya ordenados y los funde, haciendo que la zona de memoria que ambos ocupan pase a estar completamente ordenada. Este gráfico muestra cómo se fundirían, paso a paso, dos vectores, *a* y *b* para formar un nuevo vector *c*. Necesitamos tres índices, *i*, *j* y *k*, uno para cada vector:



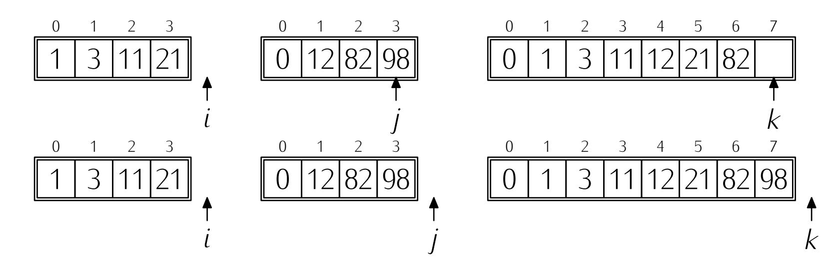
Inicialmente, los tres índices valen 0. Ahora comparamos *a*[*i*] con *b*[*j*], seleccionamos el menor y almacenamos el valor en *c*[*k*]. Es necesario incrementar *i* si escogimos un elemento de *a* y *j* si lo escogimos de *b*. En cualquier caso, hemos de incrementar también la variable *k*:



El proceso se repite hasta que alguno de los dos primeros índices, *i* o *j*, se «sale» del vector correspondiente, tal y como ilustra esta secuencia de imágenes:



Ahora, basta con copiar los últimos elementos del otro vector al final de *c*:



Un último paso del proceso de fusión debería copiar los elementos de *c* en *a* y *b*, que en realidad son fragmentos contiguos de un mismo vector.

Vamos a por los detalles de implementación. No trabajamos con dos vectores independientes, sino con un sólo vector en el que se marcan «subvectores» con pares de índices.

```
void merge(int v[], int inicio1, int final1, int inicio2, int final2)
{
   int i, j, k;
   int c[final2-inicio1+1]; // Vector de talla determinada en tiempo de ejecución.

   i = inicio1;
   j = inicio2;
   k = 0;

   while (i<=final1 && j<=final2)
      if (v[i] < v[j])
         c[k++] = v[i++];
      else
         c[k++] = v[j++];

```

```
   while (i<=final1)
      c[k++] = v[i++];

   while (j<=final2)
      c[k++] = v[j++];

   for (k=0; k<final2-inicio1+1; k++)
      v[inicio1+k] = c[k];
}
```

El último paso del procedimiento se encarga de copiar los elementos de *c* en el vector original.

Ya está. Bueno, aún podemos efectuar una mejora para reducir el número de parámetros: fíjate en que *inicio*2 siempre es igual a *final*1+1. Podemos prescindir de uno de los dos parámetros:

```
void merge(int v[], int inicio1, int final1, int final2)
{
   int i, j, k;
   int c[final2-inicio1+1];

   i = inicio1;
   j = final1+1;
   k = 0;

   while (i<=final1 && j<=final2)
      if (v[i] < v[j])
         c[k++] = v[i++];
      else
         c[k++] = v[j++];

   while (i<=final1)
      c[k++] = v[i++];

   while (j<=final2)
      c[k++] = v[j++];

   for (k=0; k<final2-inicio1+1; k++)
      v[inicio1+k] = c[k];
}
```

Veamos cómo quedaría un programa completo que use *mergesort*:

ordena.c ordena.c

```
#include <stdio.h>

#define TALLA 100

void merge(int v[], int inicio1, int final1, int final2)
{
   int i, j, k;
   int c[final2-inicio1+1];

   i = inicio1;
   j = final1+1;
   k = 0;

   while (i<=final1 && j<=final2)
      if (v[i] < v[j])
```

```
         c[k++] = v[i++];
      else
         c[k++] = v[j++];

   while (i<=final1)
      c[k++] = v[i++];

   while (j<=final2)
      c[k++] = v[j++];

   for (k=0; k<final2-inicio1+1; k++)
      v[inicio1+k] = c[k];
}

void mergesort(int v[], int inicio, int final)
{
   if (final - inicio == 0)
      return;
   else {
      mergesort ( v, inicio, (inicio+final) / 2 );
      mergesort ( v, (inicio+final) / 2 + 1, final );
      merge( v, inicio, (inicio+final) / 2, final );
   }
}

int main(void)
{
   int mivector[TALLA];
   int i, talla;

   talla = 0;
   for (i=0; i<TALLA; i++) {
      printf("Introduce elemento %d (negativo para acabar): ", i);
      scanf("%d", &mivector[i]);
      if (mivector[i] < 0)
         break;
      talla++;
   }

   mergesort(mivector, 0, talla-1);

   printf("Vector ordenado:\n");
   for (i=0; i<talla; i++)
      printf("%d ", mivector[i]);
   printf("\n");
   return 0;
}
```

He aquí una ejecución del programa:

```
Introduce elemento 0 (negativo para acabar): 3
Introduce elemento 1 (negativo para acabar): 53
Introduce elemento 2 (negativo para acabar): 32
Introduce elemento 3 (negativo para acabar): 34
Introduce elemento 4 (negativo para acabar): 64
Introduce elemento 5 (negativo para acabar): 3
Introduce elemento 6 (negativo para acabar): 4
Introduce elemento 7 (negativo para acabar): 6
Introduce elemento 8 (negativo para acabar): 7
```

```
Introduce elemento 9 (negativo para acabar): -1↲
Vector ordenado:
3 3 4 6 7 32 34 53 64
```

Mergesort y el estilo C

Los programadores C tienden a escribir los programas de una forma muy compacta. Estudia esta nueva versión de la función *merge*:

```
void merge(int v[], int inicio1, int final1, int final2)
{
  int i, j, k;
  int c[final2-inicio1+1];

  for (i=inicio1, j=final1+1, k=0; i<=final1 && j<=final2;  )
    c[k++] = (v[i] < v[j]) ? v[i++] : v[j++];
  while (i<=final1) c[k++] = v[i++];
  while (j<=final2) c[k++] = v[j++];
  for (k=0; k<final2-inicio1+1; k++) v[inicio1+k] = c[k];
}
```

Observa que los bucles **for** aceptan más de una inicialización (separándolas por comas) y permiten que alguno de sus elementos esté en blanco (en el primer **for** la acción de incremento del índice está en blanco). No te sugerimos que hagas tú lo mismo: te prevenimos para que estés preparado cuando te enfrentes a la lectura de programas C escritos por otros.

También vale la pena apreciar el uso del operador ternario para evitar una estructura condicional **if-else** que en sus dos bloques asigna un valor a la misma celda del vector. Es una práctica frecuente y da lugar, una vez acostumbrado, a programas bastante legibles.

3.6.2. Recursión indirecta y declaración anticipada

C debe conocer la cabecera de una función antes de que sea llamada, es decir, debe conocer el tipo de retorno y el número y tipo de sus parámetros. Normalmente ello no plantea ningún problema: basta con definir la función antes de su uso, pero no siempre es posible. Imagina que una función f necesita llamar a una función g y que g, a su vez, necesita llamar a f (recursión indirecta). ¿Cuál ponemos delante? La solución es fácil: da igual, la que quieras, pero debes hacer una *declaración anticipada* de la función que defines en segundo lugar. La declaración anticipada no incluye el cuerpo de la función: consiste en la declaración del tipo de retorno, identificador de función y lista de parámetros con su tipo, es decir, es un *prototipo* o *perfil* de la función en cuestión.

Estudia este ejemplo[6]:

```
int impar(int a);

int par(int a)
{
  if (a==0)
    return 1;
  else
    return (impar(a-1));
```

[6] El ejemplo es meramente ilustrativo: hay formas mucho más eficientes de saber si un número es par o impar.

```
}

int impar(int a)
{
   if (a==0)
      return 0;
   else
      return (par(a-1));
}
```

La primera línea es una declaración anticipada de la función *impar*, pues se usa antes de haber sido definida. Con la declaración anticipada hemos «adelantado» la información acerca de qué tipo de valores aceptará y devolverá la función.

. . . . EJERCICIOS . . . .

▶ 211 Dibuja el estado de la pila cuando se llega al caso base en la llamada recursiva *impar*(7).

. . . .

La declaración anticipada resulta necesaria para programas con recursión indirecta, pero también la encontrarás (o usarás) en programas sin recursión. A veces conviene definir funciones en un orden que facilite la lectura del programa, y es fácil que se defina una función después de su primer uso. Pongamos por caso el programa `ordena.c` en el que hemos implementado el método de ordenación por fusión: puede que resulte más legible definir primero *mergesort* y después *merge* pues, a fin de cuentas, las hemos desarrollado en ese orden. De definirlas así, necesitaríamos declarar anticipadamente *merge*:

ordena.c

```
#include <stdio.h>

#define TALLA 100

void merge(int v[], int inicio1, int final1, int final2) ; // Declaración anticipada.

void mergesort(int v[], int inicio, int final)
{
   if (final - inicio == 0)
      return;
   else {
      mergesort ( v, inicio, (inicio+final) / 2 );
      mergesort ( v, (inicio+final) / 2 + 1, final );
      merge( v, inicio, (inicio+final) / 2, final ); // Podemos usarla: se ha declarado antes.
   }
}

void merge(int v[], int inicio1, int final1, int final2) // Y ahora se define.
{
...
```

3.7. Macros

El preprocesador permite definir un tipo especial de funciones que, en el fondo, no lo son: las *macros*. Una macro tiene parámetros y se usa como una función cualquiera, pero las llamadas no se traducen en verdaderas llamadas a función. Ahora verás por qué.

Vamos con un ejemplo:

Prototipo por defecto y declaración anticipada

Si usas una función antes de definirla y no has preparado una declaración anticipada, C deduce el tipo de cada parámetro a partir de la forma en la que se le invoca. Este truco funciona a veces, pero es frecuente que sea fuente de problemas. Considera este ejemplo:

```
1 int f (int y)
2 {
3     return 1 + g(y);
4 }
5
6 float g(float x)
7 {
8     return x*x;
9 }
```

En la línea 3 se usa *g* y aún no se ha definido. Por la forma de uso, el compilador deduce que su perfile es **int** *g*(**int** *x*). Pero, al ver la definición, detecta un conflicto.

El problema se soluciona alterando el orden de definición de las funciones o, si se prefiere, mediante una declaración anticipada:

```
float g(float x);

int f (int y)
{
    return 1 + g(y);
}

float g(float x)
{
    return x*x;
}
```

```
#define CUADRADO(x) x*x
```

La directiva con la que se define una macro es **#define**, la misma con la que declarábamos constantes. La diferencia está en que la macro lleva uno o más parámetros (separados por comas) encerrados entre paréntesis. Este programa define y usa la macro CUADRADO:

```
#include <stdio.h>

#define CUADRADO(x) x*x

int main (void)
{
  printf("El␣cuadrado␣de␣%d␣es␣%d\n", 2, CUADRADO(2));
  return 0;
}
```

El compilador no llega a ver nunca la llamada a CUADRADO. La razón es que el preprocesador la sustituye por su cuerpo, consiguiendo que el compilador vea esta otra versión del programa:

```
#include <stdio.h>


```

```
int main (void)
{
  printf("El␣cuadrado␣de␣%d␣es␣%d\n", 2, 2*2);
  return 0;
}
```

Las macros presentan algunas ventajas frente a las funciones:

- Por regla general, son más rápidas que las funciones, pues al no implicar una llamada a función en tiempo de ejecución nos ahorramos la copia de argumentos en pila y el salto/retorno a otro lugar del programa.
- No obligan a dar información de tipo acerca de los parámetros ni del valor de retorno. Por ejemplo, esta macro devuelve el máximo de dos números, sin importar que sean enteros o flotantes:

```
#define MAXIMO(A, B) ((A > B) ? A : B)
```

Pero tienen serios inconvenientes:

- La definición de la macro debe ocupar, en principio, una sola línea. Si ocupa más de una línea, hemos de finalizar todas menos la última con el carácter «\» justo antes del salto de línea. Incómodo.
- No puedes definir variables locales.[7]
- No admiten recursión.
- *Son peligrosísimas*. ¿Qué crees que muestra por pantalla este programa?:

```
#include <stdio.h>

#define CUADRADO(x) x*x

int main (void)
{
  printf("El␣cuadrado␣de␣6␣es␣%d\n", CUADRADO(3+3));
  return 0;
}
```

¿36?, es decir, ¿el cuadrado de 6? Pues no es eso lo que obtienes, sino 15. ¿Por qué? El preprocesador sustituye el fragmento CUADRADO(3+3) por... ¡3+3*3+3!

El resultado es, efectivamente, 15, y no el que esperábamos. Puedes evitar este problema usando paréntesis:

```
#include <stdio.h>

#define CUADRADO(x) (x)*(x)

main (void)
{
  printf("El␣cuadrado␣de␣6␣es␣%d\n", CUADRADO(3+3));
  return 0;
}
```

[7]No del todo cierto, pero no entraremos en detalles.

Ahora el fragmento `CUADRADO(3+3)` se sustituye por `(3+3)*(3+3)`, que es lo que esperamos. Otro problema resuelto.

No te fíes. Ya te hemos dicho que las macros son peligrosas. Sigue estando mal. ¿Qué esperas que calcule `1.0/CUADRADO(3+3)`?, ¿el valor de 1/36, es decir, 0.02777...? Te equivocas. La expresión `1.0/CUADRADO(3+3)` se convierte en `1.0/(3+3)*(3+3)`, que es $1/6 \cdot 6$, o sea, 1, no 1/36.

La solución pasa por añadir nuevos paréntesis:

```
#include <stdio.h>

#define CUADRADO(x) ((x)*(x))

...
```

¿Ahora sí? La expresión `1.0/CUADRADO(3+3)` se convierte en `1.0/((3+3)*(3+3))`, que es 1/36. Pero todavía hay un problema: si ejecutamos este fragmento de código:

```
i = 3;
z = CUADRADO(i++);
```

la variable se incrementa 2 veces, y no una sóla. Ten en cuenta que el compilador traduce lo que «ve», y «ve» esto:

```
i = 3;
z = ((i++)*(i++));
```

Y este problema no se puede solucionar.

¡Recuerda! Si usas macros, toda precaución es poca.

EJERCICIOS

▶ 212 Diseña una macro que calcule la tangente de una cantidad de radianes. Puedes usar las funciones *sin* y *cos* de `math.h`, pero ninguna otra.

▶ 213 Diseña una macro que devuelva el mínimo de dos números, sin importar si son enteros o flotantes.

▶ 214 Diseña una macro que calcule el valor absoluto de un número, sin importar si es entero o flotante.

▶ 215 Diseña una macro que decremente una variable entera si y sólo si es positiva. La macro devolverá el valor ya decrementado o inalterado, según convenga.

3.8. Otras cuestiones acerca de las funciones

3.8.1. Funciones **inline**

Los inconvenientes de las macros desaconsejan su uso. Lenguajes como C++ dan soporte a las macros sólo por compatibilidad con C, pero ofrecen alternativas mejores. Por ejemplo, puedes definir funciones **inline**. Una función **inline** es como cualquier otra función, sólo que las llamadas a ella se gestionan como las llamadas a macros: se sustituye la llamada por el código que se ejecutaría en ese caso, o sea, por el cuerpo de la función con los valores que se suministren para los parámetros. Las funciones **inline** presentan muchas ventajas frente a la macros. Entre ellas, la posibilidad de utilizar variables locales o la no necesidad de utilizar paréntesis alrededor de toda aparición de un parámetro.

Las funciones **inline** son tan útiles que compiladores como `gcc` las integran desde hace años como extensión propia del lenguaje C y han pasado a formar parte del lenguaje C99. Al compilar un programa C99 como éste:

```
#include <stdio.h>

inline int doble(int a)
{
  return a * 2;
}

int main(void)
{
  int i;

  for (i=0; i<10; i++)
    printf("%d\n", doble(i+1) );

  return 0;
}
```

no se genera código de máquina con 10 llamadas a la función *doble*. El código de máquina que se genera es virtualmente idéntico al que se genera para este otro programa equivalente:

```
#include <stdio.h>

int main(void)
{
  int i;

  for (i=0; i<10; i++)
    printf("%d\n", ((i+1) * 2) );

  return 0;
}
```

Hay ocasiones, no obstante, en las que el compilador no puede efectuar la sustitución de la llamada a función por su cuerpo. Si la función es recursiva, por ejemplo, la sustitución es imposible. Pero aunque no sea recursiva, el compilador puede juzgar que una función es excesivamente larga o compleja para que compense efectuar la sustitución. Cuando se declara una función como **inline**, sólo se está sugiriendo al compilador que efectúe la sustitución, pero éste tiene la última palabra sobre si habrá o no una verdadera llamada a función.

3.8.2. Variables locales **static**

Hay un tipo especial de variable local: las variables **static**. Una variable **static** es invisible fuera de la función, como cualquier otra variable local, pero recuerda su valor entre diferentes ejecuciones de la función en la que se declara.

Veamos un ejemplo:

```
#include <stdio.h>

int turno(void)
{
  static int contador = 0;

  return contador++;
}

int main(void)
```

```
{
   int i;

   for (i=0; i<10; i++)
      printf("%d\n", turno());
   return 0;
}
```

Si ejecutas el programa aparecerán por pantalla los números del 0 al 9. Con cada llamada, *contador* devuelve su valor y se incrementa en una unidad, sin olvidar su valor entre llamada y llamada.

La inicialización de las variables **static** es opcional: el compilador asegura que empiezan valiendo 0.

Vamos a volver a escribir el programa que presentamos en el ejercicio 169 para generar números primos consecutivos. Esta vez, vamos a hacerlo sin usar una variable global que recuerde el valor del último primo generado. Usaremos en su lugar una variable local **static**:

primos.c   primos.c

```
#include <stdio.h>

int siguienteprimo(void)
{
   static int ultimoprimo = 0;
   int esprimo;
   int i;

   do {
      ultimoprimo++;
      esprimo = 1;
      for (i=2; i<ultimoprimo/2; i++)
         if (ultimoprimo % i == 0) {
            esprimo = 0;
            break;
         }
   } while (!esprimo);
   return ultimoprimo;
}

int main(void)
{
   int i;

   printf("Los␣10␣primeros␣números␣primos\n");
   for (i=0; i<10; i++)
      printf("%d\n", siguienteprimo());
   return 0;
}
```

Mucho mejor. Si puedes evitar el uso de variables globales, evítalo. Las variables locales **static** pueden ser la solución en bastantes casos.

3.8.3. Paso de funciones como parámetros

Hay un tipo de parámetro especial que puedes pasar a una función: ¡otra función!

Veamos un ejemplo. En este fragmento de programa se definen sendas funciones C que aproximan numéricamente la integral definida en un intervalo para las funciones matemáticas $f(x) = x^2$ y $f(x) = x^3$, respectivamente:

```
float integra_cuadrado(float a, float b, int n)
{
  int i;
  float s, x;

  s = 0.0;
  x = a;
  for (i=0; i<n; i++) {
    s += x*x * (b-a)/n;
    x += (b-a)/n;
  }
  return s;
}

float integra_cubo(float a, float b, int n)
{
  int i;
  float s, x;

  s = 0.0;
  x = a;
  for (i=0; i<n; i++) {
    s += x*x*x * (b-a)/n;
    x += (b-a)/n;
  }
  return s;
}
```

Las dos funciones que hemos definido son básicamente iguales. Sólo difieren en su identificador y en la función matemática que integran. ¿No sería mejor disponer de una única función C, digamos *integra*, a la que suministremos como parámetro la función matemática que queremos integrar? C lo permite:

```
float integra(float a, float b, int n, float (*f)(float))
{
  int i;
  float s, x;

  s = 0.0;
  x = a;
  for (i=0; i<n; i++) {
    s += f(x) * (b-a)/n;
    x += (b-a)/n;
  }
  return s;
}
```

Hemos declarado un cuarto parámetro que es de tipo *puntero a función*. Cuando llamamos a *integra*, el cuarto parámetro puede ser el identificador de una función que reciba un **float** y devuelva un **float**:

integra.c integra.c

```
#include <stdio.h>

float integra(float a, float b, int n, float (*f)(float))
{
  int i;
  float s, x;

  s = 0.0;
```

```
    x = a;
    for (i=0; i<n; i++) {
        s += f(x) * (b-a)/n;
        x += (b-a)/n;
    }
    return s;
}

float cuadrado(float x)
{
    return x*x;
}

float cubo(float x)
{
    return x*x*x;
}

int main(void)
{
    printf("Integral 1: %f\n", integra(0.0, 1.0, 10, cuadrado));
    printf("Integral 2: %f\n", integra(0.0, 1.0, 10, cubo));
    return 0;
}
```

La forma en que se declara un parámetro del tipo «puntero a función» resulta un tanto complicada. En nuestro caso, lo hemos declarado así: **float** (*f)(**float**). El primer **float** indica que la función devuelve un valor de ese tipo. El (*f) indica que el parámetro *f* es un puntero a función. Y el **float** entre paréntesis indica que la función trabaja con un parámetro de tipo **float**. Si hubiésemos necesitado trabajar con una función que recibe un **float** y un **int**, hubiésemos escrito **float** (*f)(**float**, **int**) en la declaración del parámetro.

EJERCICIOS

▶ 216 ¿Puedes usar la función *integra* para calcular la integral definida de la función matemática sin(x)? ¿Cómo?

▶ 217 Diseña una función C capaz de calcular

$$\sum_{i=a}^{b} f(i),$$

siendo f una función matemática cualquiera que recibe un entero y devuelve un entero.

▶ 218 Diseña una función C capaz de calcular

$$\sum_{i=a}^{b} \sum_{j=c}^{d} f(i,j),$$

siendo f una función matemática cualquiera que recibe dos enteros y devuelve un entero.

3.9. Módulos, bibliotecas y unidades de compilación

Cuando te enfrentas a la escritura de un programa largo, individualmente o en equipo, te resultará virtualmente imposible escribirlo en un único fichero de texto. Resulta más práctico agrupar diferentes partes del programa en ficheros independientes. Cada fichero

puede, por ejemplo, agrupar las funciones, registros y constantes propias de cierto tipo de cálculos.

Proceder así tiene varias ventajas:

- Mejora la legibilidad del código (cada fichero es relativamente breve y agrupa temáticamente las funciones, registros y constantes).
- La compilación es más rápida (cuando se modifica un fichero, sólo es necesario compilar ese fichero).
- Y, quizá lo más importante, permite reutilizar código. Es un beneficio a medio y largo plazo. Si, por ejemplo, te dedicas a programar videojuegos tridimensionales, verás que todos ellos comparten ciertas constantes, registros y funciones definidas por tí o por otros programadores: tipos de datos para modelar puntos, polígonos, texturas, etcétera; funciones que los manipulan, visualizan, leen/escriben en disco, etcétera. Puedes definir estos elementos en un fichero y utilizarlo en cuantos programas desees. Alternativamente, podrías copiar-y-pegar las funciones, constantes y registros que uno necesita en cada programa, pero no es conveniente en absoluto: corregir un error en una función obligaría a editar todos los programas en los que se pegó; por contra, si está en un solo fichero, basta con corregir la definición una sola vez.

C permite escribir un programa como una colección de *unidades de compilación*. El concepto es similar al de los módulos Python: cada unidad agrupa definiciones de variables, tipos, constantes y funciones orientados a resolver cierto tipo de problemas. Puedes compilar independientemente cada unidad de compilación (de ahí el nombre) de modo que el compilador genere un fichero binario para cada una de ellas. El enlazador se encarga de unir en una última etapa todas las unidades compiladas para crear un único fichero ejecutable.

Lo mejor será que aprendamos sobre unidades de compilación escribiendo una muy sencilla: un módulo en el que se define una función que calcula el máximo de dos número enteros. El fichero que corresponde a esta unidad de compilación se llamará `extremos.c`. He aquí su contenido:

extremos.c — extremos.c

```
int maximo(int a, int b)
{
  if (a > b)
    return a;
  else
    return b;
}
```

El programa principal se escribirá en otro fichero llamado `principal.c`. Dicho programa llamará a la función *maximo*:

principal_1.c — ⚡ principal.c ⚡

```
#include <stdio.h>

int main(void)
{
  int x, y;

  printf("Dame␣un␣número:␣");
  scanf("%d", &x);
  printf("Dame␣otro:␣");
  scanf("%d", &y);
  printf("El␣máximo␣es␣%d\n", maximo(x, y));
```

```
    return 0;
}
```

Hemos marcado el programa como incorrecto. ¿Por qué? Verás, estamos usando una función, *maximo*, que no está definida en el fichero `principal.c`. ¿Cómo sabe el compilador cuántos parámetros recibe dicha función?, ¿y el tipo de cada parámetro?, ¿y el tipo del valor de retorno? El compilador se ve obligado a generar código de máquina para llamar a una función de la que no sabe nada. Mala cosa.

¿Cómo se resuelve el problema? Puedes *declarar* la función sin *definirla*, es decir, puedes declarar el aspecto de su cabecera (lo que denominamos su *prototipo*) e indicar que es una función definida *externamente*:

principal_2.c — `principal.c`

```
#include <stdio.h>

extern int maximo(int a, int b);

int main(void)
{
    int x, y;

    printf("Dame un número: ");
    scanf("%d", &x);
    printf("Dame otro: ");
    scanf("%d", &y);
    printf("El máximo es %d\n", maximo(x, y));
    return 0;
}
```

El prototipo contiene toda la información útil para efectuar la llamada a la función, pero no contiene su cuerpo: la cabecera acaba con un punto y coma. Fíjate en que la declaración del prototipo de la función *maximo* empieza con la palabra clave **extern**. Con ella se indica al compilador que *maximo* está definida en algún módulo «externo». También puedes indicar con **extern** que una variable se define en otro módulo.

Puedes compilar el programa así:

```
$ gcc extremos.c -c
$ gcc principal.c -c
$ gcc principal.o extremos.o -o principal
```

La compilación necesita tres pasos: uno por cada unidad de compilación y otro para enlazar.

1. El primer paso (`gcc extremos.c -c`) traduce a código de máquina el fichero o unidad de compilación `extremos.c`. La opción `-c` indica al compilador que `extremos.c` es un módulo y no define a la función *main*. El resultado de la compilación se deja en un fichero llamado `extremos.o`. La extensión «`.o`» abrevia el término «object code», es decir, «código objeto». Los ficheros con extensión «`.o`» contienen el código de máquina de nuestras funciones[8], pero no es directamente ejecutable.

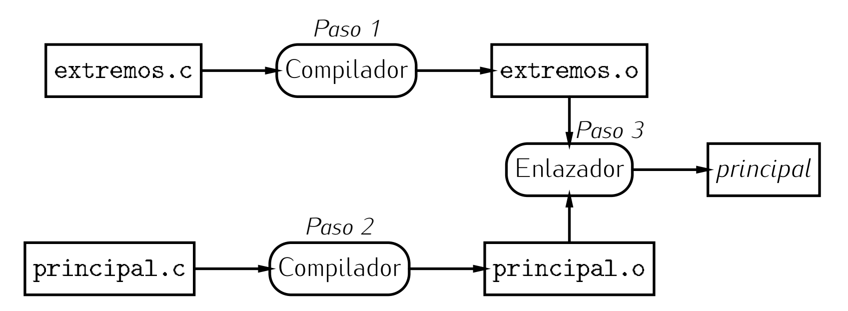
2. El segundo paso (`gcc principal.c -c`) es similar al primero y genera el fichero `principal.o` a partir de `principal.c`.

[8]... pero no sólo eso: también contienen otra información, como la denominada tabla de símbolos.

3. El tercer paso (`gcc principal.o extremos.o -o principal`) es especial. El compilador recibe dos ficheros con extensión «`.o`» y genera un único fichero ejecutable, llamado *principal*. Este último paso se encarga de *enlazar* las dos unidades compiladas para generar el fichero ejecutable.

   Por enlazar entendemos que las llamadas a funciones cuyo código de máquina era desconocido (estaba en otra unidad de compilación) se traduzcan en «saltos» a las direcciones en las que se encuentran los subprogramas de código máquina correspondientes (y que ahora se conocen).

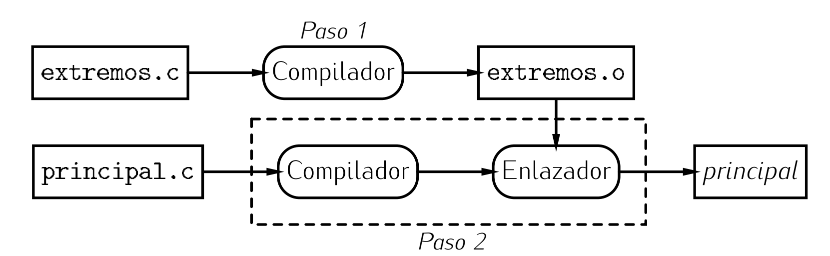
Aquí tienes un diagrama que ilustra el proceso:



Puedes ahorrarte un paso fundiendo los dos últimos en uno sólo. Así:

```
$ gcc extremos.c -c ↵
$ gcc principal.c extremos.o -o principal ↵
```

Este diagrama muestra todos los pasos del proceso a los que aludimos:

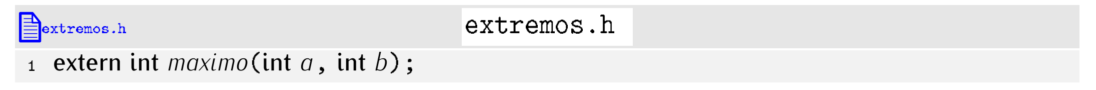


Para conseguir un programa ejecutable es necesario que uno de los módulos (¡pero sólo uno de ellos!) defina una función *main*. Si ningún módulo define *main* o si *main* se define en más de un módulo, el enlazador protestará y no generará fichero ejecutable alguno.

3.9.1. Declaración de prototipos en cabeceras

Hemos resuelto el problema de gestionar diferentes unidades de compilación, pero la solución de tener que declarar el prototipo de cada función en toda unidad de compilación que la usa no es muy buena. Hay una mejor: definir un *fichero de cabecera*. Los ficheros de cabecera agrupan las declaraciones de funciones (y cualquier otro elemento) definidos en un módulo. Las cabeceras son ficheros con extensión «`.h`» (es un convenio: la «h» es abreviatura de «header»).

Nuestra cabecera será este fichero:




```
extern int maximo(int a, int b);
```

Para incluir la cabecera en nuestro programa, escribiremos una nueva directiva **#include**:

Documentación y cabeceras

Es importante que documentes bien los ficheros de cabecera, pues es frecuente que los programadores que usen tu módulo lo consulten para hacerse una idea de qué ofrece.

Nuestro módulo podría haberse documentado así:

extremos.h

```c
/*******************************************************
 * Módulo: extremos
 *
 * Propósito: funciones para cálculo de valores máximos
 * y mínimos.
 *
 * Autor: A. U. Thor.
 *
 * Fecha: 12 de enero de 1997
 *
 * Estado: Incompleto. Falta la función minimo.
 *******************************************************/

extern int maximo(int a, int b);
  /* Calcula el máximo de dos número enteros a y b. */
```

¿Y por qué los programadores no miran directamente el fichero `.c` en lugar del `.h` cuando quieren consultar algo? Por varias razones. Una de ellas es que, posiblemente, el `.c` no esté accesible. Si el módulo es un producto comercial, probablemente sólo les hayan vendido el módulo ya compilado (el fichero `.o`) y el fichero de cabecera. Pero incluso si se tiene acceso al `.c`, puede ser preferible ver el `.h`. El fichero `.c` puede estar plagado de detalles de implementación, funciones auxiliares, variables para uso interno, etc., que hacen engorrosa su lectura. El fichero de cabecera contiene una somera declaración de cada uno de los elementos del módulo que se «publican» para su uso en otros módulos o programas, así que es una especie de resumen del `.c`.

principal.c

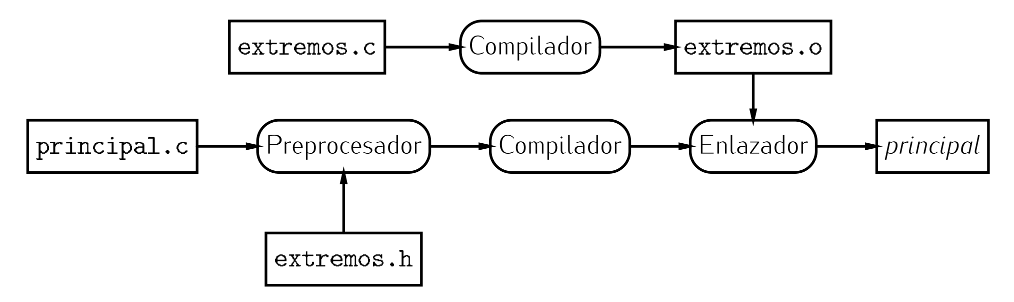
```c
#include <stdio.h>
#include "extremos.h"

int main(void)
{
  int x, y;

  printf("Dame un número: ");
  scanf("%d", &x);
  printf("Dame otro: ");
  scanf("%d", &y);
  printf("El máximo es %d\n", maximo(x, y));
  return 0;
}
```

La única diferencia con respecto a otros **#include** que ya hemos usado estriba en el uso de comillas dobles para encerrar el nombre del fichero, en lugar de los caracteres «<» y «>». Con ello indicamos al preprocesador que el fichero `extremos.h` se encuentra en nuestro directorio activo. El preprocesador se limita a sustituir la línea en la que aparece **#include** `"extremos.h"` por el contenido del fichero. En un ejemplo tan sencillo no hemos ganado mucho, pero si el módulo `extremos.o` contuviera muchas funciones, con sólo una línea habríamos conseguido «importarlas» todas.

Aquí tienes una actualización del gráfico que muestra el proceso completo de compilación:



3.9.2. Declaración de variables en cabeceras

No sólo puedes declarar funciones en los ficheros de cabecera. También puedes definir constantes, variables y registros.

Poco hay que decir sobre las constantes. Basta con que las definas con **#define** en el fichero de cabecera. Las variables, sin embargo, sí plantean un problema. Este módulo, por ejemplo, declara una variable entera en `mimodulo.c`:

`mimodulo.c`

```
int variable;
```

Si deseamos que otras unidades de compilación puedan acceder a esa variable, tendremos que incluir su declaración en la cabecera. ¿Cómo? Una primera idea es poner, directamente, la declaración así:

`mimodulo.h`

```
int variable;
```

Pero es incorrecta. El problema radica en que cuando incluyamos la cabecera `mimodulo.h` en nuestro programa, se insertará la línea **int** *variable*;, sin más, así que se estará definiendo una *nueva* variable con el mismo identificador que otra. Y declarar dos variables con el mismo identificador es un error.

Quien detecta el error es el enlazador: cuando vaya a generar el programa ejecutable, encontrará que hay dos objetos que tienen el mismo identificador, y eso está prohibido. La solución es sencilla: preceder la declaración de variable en la cabecera `mimodulo.h` con la palabra reservada **extern**:

`mimodulo.h`

```
extern int variable;
```

De ese modo, cuando se compila un programa que incluye a `mimodulo.h`, el compilador sabe que *variable* es de tipo **int** y que está definida en alguna unidad de compilación, por lo que no la crea por segunda vez.

3.9.3. Declaración de registros en cabeceras

Finalmente, puedes declarar también registros en las cabeceras. Como los programas que construiremos son sencillos, no se planteará problema alguno con la definición de registros: basta con que pongas su declaración en la cabecera, sin más. Pero si tu programa incluye dos cabeceras que, a su vez, incluyen ambas a una tercera donde se definen constantes o registros, puedes tener problemas. Un ejemplo ilustrará mejor el tipo de dificultades al que nos enfrentamos. Supongamos que un fichero `a.h` define un registro:

Bibliotecas

Ya has usado funciones y datos predefinidos, como las funciones y las constantes matemáticas. Hemos hablado entonces del uso de la *biblioteca* matemática. ¿Por qué «biblioteca» y no «módulo»? Una biblioteca es más que un módulo: es un conjunto de módulos.

Cuando se tiene una pléyade de ficheros con extensión «.o», conviene empaquetarlos en uno solo con extensión «.a» (por «archive»). Los ficheros con extensión «.a» son similares a los ficheros con extensión «.tar»: meras colecciones de ficheros. De hecho, «tar» (*tape archiver*) es una evolución de «.ar» (por «archiver»), el programa con el que se manipulan los ficheros con extensión «.a».

La biblioteca matemática, por ejemplo, agrupa un montón de módulos. En un sistema Linux se encuentra en el fichero /usr/lib/libm.a y puedes consultar su contenido con esta orden:

```
$ ar tvf /usr/lib/libm.a
rw-r--r-- 0/0  29212 Sep  9 18:17 2002 k_standard.o
rw-r--r-- 0/0   8968 Sep  9 18:17 2002 s_lib_version.o
rw-r--r-- 0/0   9360 Sep  9 18:17 2002 s_matherr.o
rw-r--r-- 0/0   8940 Sep  9 18:17 2002 s_signgam.o
⋮
rw-r--r-- 0/0   1152 Sep  9 18:17 2002 slowexp.o
rw-r--r-- 0/0   1152 Sep  9 18:17 2002 slowpow.o
```

Como puedes ver, hay varios ficheros con extensión «.o» en su interior. (Sólo te mostramos el principio y el final del resultado de la llamada, pues hay un total de ¡395 ficheros!)

Cuando usas la biblioteca matemática compilas así:

```
$ gcc programa.c -lm -o programa
```

o, equivalentemente, así:

```
$ gcc programa.c /usr/lib/libm.a -o programa
```

En el segundo caso hacemos explícito el nombre de la biblioteca en la que se encuentran las funciones matemáticas. El enlazador no sólo sabe tratar ficheros con extensión «.o»: también sabe buscarlos en los de extensión «.a».

En cualquier caso, sigue siendo necesario que las unidades de compilación conozcan el perfil de las funciones que usan y están definidas en otros módulos o bibliotecas. Por eso incluímos, cuando conviene, el fichero math.h en nuestros programas.

Hay infinidad de bibliotecas que agrupan módulos con utilidades para diferentes campos de aplicación: resolución de problemas matemáticos, diseño de videojuegos, reproducción de música, etc. Algunas son código abierto, en cuyo caso se distribuyen con los ficheros de extensión «.c», los ficheros de extensión «.h» y alguna utilidad para facilitar la compilación (un *makefile*). Cuando son comerciales es frecuente que se mantenga el código fuente en privado. En tal caso, se distribuye el fichero con extensión «.a» (o una colección de ficheros con extensión «.o») y uno o más ficheros con extensión «.h».

a.h

```c
// Cabecera a.h
struct A {
  int a;
};
// Fin de cabecera a.h
```

Ahora, los ficheros b.h y c.h incluyen a a.h y declaran la existencia de sendas funciones:

b.h

```
// Cabecera b.h
#include "a.h"

int funcion_de_b_punto_h(int x);
// Fin de cabecera b.h
```

c.h

```
// Cabecera c.h
#include "a.h"

int funcion_de_c_punto_h(int x);
// Fin de cabecera c.h
```

Y, finalmente, nuestro programa incluye tanto a b.h como a c.h:

programa.c

```
#include <stdio.h>

#include "b.h"

#include "c.h"

int main(void)
{
  ...
}
```

El resultado es que el a.h acaba quedando incluido ¡dos veces! Tras el paso de programa.c por el preprocesador, el compilador se enfrenta, a este texto:

programa.c

```
#include <stdio.h>

// Cabecera b.h.
// Cabecera a.h.
struct A {
  int a;
};
// Fin de cabecera a.h.

int funcion_de_b_punto_h(int x);
// Fin de cabecera b.h.

// Cabecera c.h.
// Cabecera a.h.
struct A {
  int a;
};
// Fin de cabecera a.h.

int funcion_de_c_punto_h(int x);
// Fin de cabecera c.h.

int main(void)
{
  ...
}
```

El compilador encuentra, por tanto, la definición de **struct** A por duplicado, y nos avisa del «error». No importa que las dos veces se declare de la misma forma: C lo considera ilegal. El problema puede resolverse reescribiendo `a.h` (y, en general, cualquier fichero cabecera) así:

```
1 // Cabecera de a.h
2 #ifndef A_H
3 #define A_H
4
5 struct A {
6   int a;
7 };
8
9 #endif
10 // Fin de cabecera de a.h
```

Las directivas **#ifndef**/**#endif** marcan una zona de «código condicional». Se interpretan así: «si la constante A_H no está definida, entonces incluye el fragmento hasta el **#endif**, en caso contrario, sáltate el texto hasta el **#endif**». O sea, el compilador verá o no lo que hay entre las líneas 3 y 8 en función de si existe o no una determinada constante. No debes confundir estas directivas con una sentencia **if**: no lo son. La sentencia **if** permite ejecutar o no un bloque de sentencias en función de que se cumpla o no una condición *en tiempo de ejecución*. Las directivas presentadas permiten que el compilador vea o no un fragmento arbitrario de texto en función de si existe o no una constante *en tiempo de compilación*.

Observa que lo primero que se hace en ese fragmento de programa es definir la constante A_H (línea 3). La primera vez que se incluya la cabecera `a.h` no estará aún definida A_H, así que se incluirán las líneas 3–8. Uno de los efectos será que A_H pasará a estar definida. La segunda vez que se incluya la cabecera `a.h`, A_H ya estará definida, así que el compilador no verá por segunda vez la definición de **struct** A.

El efecto final es que la definición de **struct** A sólo se ve una vez. He aquí lo que resulta de `programa.c` tras su paso por el preprocesador:

programa.c

```
#include <stdio.h>

// Cabecera b.h.
// Cabecera a.h.
struct A {
  int a;
};
// Fin de cabecera a.h.

int funcion_de_b_punto_h(int x);
// Fin de cabecera b.h.

// Cabecera c.h.
// Cabecera a.h.
// Fin de cabecera a.h.

int funcion_de_c_punto_h(int x);
// Fin de cabecera c.h.

int main(void)
{
  ...
}
```

La segunda inclusión de `a.h` no ha supuesto el copiado del texto guardado entre directivas **#ifndef/#endif**. Ingenioso, ¿no?

Capítulo 4

Estructuras de datos: memoria dinámica

La Reina se puso congestionada de furia, y, tras lanzarle una mirada felina, empezó a gritar: «¡Que le corten la cabeza! ¡Que le corten…!».

LEWIS CARROLL, *Alicia en el País de las Maravillas*.

Vimos en el capítulo 2 que los vectores de C presentaban un serio inconveniente con respecto a las listas de Python: su tamaño debía ser fijo y conocido en tiempo de compilación, es decir, no podíamos alargar o acortar los vectores para que se adaptaran al tamaño de una serie de datos *durante la ejecución del programa*. C permite una gestión dinámica de la memoria, es decir, solicitar memoria para albergar el contenido de estructuras de datos cuyo tamaño exacto no conocemos hasta que se ha iniciado la ejecución del programa. Estudiaremos aquí dos formas de superar las limitaciones de tamaño que impone el C:

- mediante vectores cuyo tamaño se fija en tiempo de ejecución,
- y mediante registros enlazados, también conocidos como *listas enlazadas* (o, simplemente, listas).

Ambas aproximaciones se basan en el uso de punteros y cada una de ellas presenta diferentes ventajas e inconvenientes.

4.1. Vectores dinámicos

Sabemos definir vectores indicando su tamaño en tiempo de compilación:

```
#define TALLA 10

int a[TALLA];
```

Pero, ¿y si no sabemos *a priori* cuántos elementos debe albergar el vector?[1] Por lo estudiado hasta el momento, podemos definir TALLA como el número más grande de elementos posible, el número de elementos para el peor de los casos. Pero, ¿y si no podemos determinar un número máximo de elementos? Aunque pudiéramos, ¿y si éste fuera tan grande que, en la práctica, supusiera un despilfarro de memoria intolerable para situaciones normales? Imagina una aplicación de agenda telefónica personal que,

[1]En la sección 3.5.3 vimos cómo definir vectores *locales* cuya talla se decide al ejecutar una función: lo que denominamos «vectores de longitud variable». Nos proponemos dos objetivos: por una parte, poder redimensionar vectores *globales*; y, por otro, vamos a permitir que un vector crezca y decrezca en tamaño cuantas veces queramos. Los «vectores de longitud variable» que estudiamos en su momento son inapropiados para cualquiera de estos dos objetivos.

por si acaso, reserva 100000 entradas en un vector. Lo más probable es que un usuario convencional no gaste más de un centenar. Estaremos desperdiciando, pues, unas 99900 celdas del vector, cada una de las cuales puede consistir en un centenar de bytes. Si todas las aplicaciones del ordenador se diseñaran así, la memoria disponible se agotaría rapidísimamente.

4.1.1. *malloc*, *free* y NULL

Afortunadamente, podemos definir, durante la ejecución del programa, vectores cuyo tamaño es exactamente el que el usuario necesita. Utilizaremos para ello dos funciones de la biblioteca estándar (disponibles incluyendo la cabecera `stdlib.h`):

- *malloc* (abreviatura de «memory allocate», que podemos traducir por «reservar memoria»): solicita un bloque de memoria del tamaño que se indique (en bytes);
- *free* (que en inglés significa «liberar»): libera memoria obtenida con *malloc*, es decir, la marca como disponible para futuras llamadas a *malloc*.

Para hacernos una idea de cómo funciona, estudiemos un ejemplo:

vector_dinamico.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6    int * a;
7    int talla, i;
8
9    printf("Número de elementos: "); scanf("%d", &talla);
10   a = malloc( talla * sizeof(int) );
11   for (i=0; i<talla; i++)
12     a[i] = i;
13   free(a);
14   a = NULL;
15
16   return 0;
17 }
```

Fíjate en cómo se ha definido el vector *a* (línea 6): como **int** * *a*, es decir, como puntero a entero. No te dejes engañar: no se trata de un puntero a *un* entero, sino de un puntero a *una secuencia* de enteros. Ambos conceptos son equivalentes en C, pues ambos son meras direcciones de memoria. La variable *a* es un *vector dinámico* de enteros, pues su memoria se obtiene dinámicamente, esto es, en tiempo de ejecución y según convenga a las necesidades. No sabemos aún cuántos enteros serán apuntados por *a*, ya que el valor de *talla* no se conocerá hasta que se ejecute el programa y se lea por teclado.

Sigamos. La línea 10 reserva memoria para *talla* enteros y guarda en *a* la dirección de memoria en la que empiezan esos enteros. La función *malloc* presenta un prototipo similar a éste:

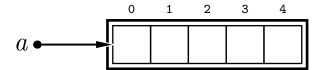
stdlib.h

```
...
void * malloc(int bytes);
...
```

Es una función que devuelve un puntero especial, del tipo de datos **void** *. ¿Qué significa **void** *? Significa «puntero a cualquier tipo de datos», o sea, «dirección de memoria»,

sin más. La función *malloc* no se usa sólo para reservar vectores dinámicos de enteros: puedes reservar con ella vectores dinámicos de cualquier tipo base. Analicemos ahora el argumento que pasamos a *malloc*. La función espera recibir como argumento un número entero: el número de bytes que queremos reservar. Si deseamos reservar *talla* valores de tipo **int**, hemos de solicitar memoria para *talla* * **sizeof**(**int**) bytes. Recuerda que **sizeof**(**int**) es la ocupación en bytes de un dato de tipo **int** (y que estamos asumiendo que es de 4).

Si el usuario decide que *talla* valga, por ejemplo, 5, se reservará un total de 20 bytes y la memoria quedará así tras ejecutar la línea 10:



Es decir, se reserva suficiente memoria para albergar 5 enteros.

Como puedes ver, las líneas 11–12 tratan a *a* como si fuera un vector de enteros cualquiera. Una vez has reservado memoria para un vector dinámico, no hay diferencia alguna entre él y un vector estático desde el punto de vista práctico. Ambos pueden indexarse (línea 12) o pasarse como argumento a funciones que admiten un vector del mismo tipo base.

Aritmética de punteros

Una curiosidad: el acceso indexado *a*[0] es equivalente a **a*. En general, *a*[*i*] es equivalente a *(*a*+*i*), es decir, ambas son formas de expresar el concepto «accede al contenido de la dirección *a* con un desplazamiento de *i* veces el tamaño del tipo base». La sentencia de asignación *a*[*i*] = *i* podría haberse escrito como *(*a*+*i*) = *i*. En C es posible sumar o restar un valor entero a un puntero. El entero se interpreta como un desplazamiento dado en unidades «tamaño del tipo base» (en el ejemplo, 4 bytes, que es el tamaño de un **int**). Es lo que se conoce por *aritmética de punteros*.

La aritmética de punteros es un punto fuerte de C, aunque también tiene sus detractores: resulta sencillo provocar accesos incorrectos a memoria si se usa mal.

Finalmente, la línea 13 del programa libera la memoria reservada y la línea 14 guarda en *a* un valor especial: NULL. La función *free* tiene un prototipo similar a éste:

stdlib.h

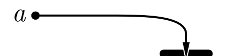
```
...
void free(void * puntero);
...
```

Como ves, *free* recibe un puntero a cualquier tipo de datos: la dirección de memoria en la que empieza un bloque previamente obtenido con una llamada a *malloc*. Lo que hace *free* es liberar ese bloque de memoria, es decir, considerar que pasa a estar disponible para otras posibles llamadas a *malloc*. Es como cerrar un fichero: si no necesito un recurso, lo libero para que otros lo puedan aprovechar.[2] Puedes aprovechar así la memoria de forma óptima.

Recuerda: *tu programa debe efectuar una llamada a free por cada llamada a malloc*. Es *muy* importante.

Conviene que después de hacer *free* asignes al puntero el valor NULL, especialmente si la variable sigue «viva» durante bastante tiempo. NULL es una constante definida en `stdlib.h`. Si un puntero vale NULL, se entiende que no apunta a un bloque de memoria. Gráficamente, un puntero que apunta a NULL se representa así:

[2]Y, como en el caso de un fichero, si no lo liberas tú explícitamente, se libera automáticamente al finalizar la ejecución del programa. Aún así, te exigimos disciplina: oblígate a liberarlo tú mismo tan pronto dejes de necesitarlo.



Liberar memoria no cambia el valor del puntero

La llamada a *free* libera la memoria apuntada por un puntero, pero no modifica el valor de la variable que se le pasa. Imagina que un bloque de memoria de 10 enteros que empieza en la dirección 1000 es apuntado por una variable *a* de tipo **int** *, es decir, imagina que *a* vale 1000. Cuando ejecutamos *free*(*a*), ese bloque se libera y pasa a estar disponible para eventuales llamadas a *malloc*, pero ¡*a* sigue valiendo 1000! ¿Por qué? Porque *a* se ha pasado a *free* por valor, no por referencia, así que *free* no tiene forma de modificar el valor de *a*. Es recomendable que asignes a *a* el valor NULL después de una llamada a *free*, pues así haces explícito que la variable *a* no apunta a nada.

Recuerda, pues, que es responsabilidad tuya y que conviene hacerlo: asigna explícitamente el valor NULL a todo puntero que no apunte a memoria reservada.

La función *malloc* puede fallar por diferentes motivos. Podemos saber cuándo ha fallado porque *malloc* lo notifica devolviendo el valor NULL. Imagina que solicitas 2 megabytes de memoria en un ordenador que sólo dispone de 1 megabyte. En tal caso, la función *malloc* devolverá el valor NULL para indicar que no pudo efectuar la reserva de memoria solicitada.

Los programas correctamente escritos deben comprobar si se pudo obtener la memoria solicitada y, en caso contrario, tratar el error.

```
a = malloc(talla * sizeof(int));
if (a == NULL) {
  printf("Error:␣no␣hay␣memoria␣suficiente\n");
}
else {
  ...
}
```

Es posible (y una forma de expresión idiomática de C) solicitar la memoria y comprobar si se pudo obtener en una única línea (presta atención al uso de paréntesis, es importante):

```
if ( (a = malloc(talla * sizeof(int))) == NULL) {
  printf("Error:␣no␣hay␣memoria␣suficiente\n");
}
else {
  ...
}
```

Nuestros programas, sin embargo, no incluirán esta comprobación. Estamos aprendiendo a programar y sacrificaremos las comprobaciones como ésta en aras de la legibilidad de los programas. Pero no lo olvides: los programas con un acabado profesional deben comprobar y tratar posibles excepciones, como la no existencia de suficiente memoria.

También puedes usar NULL para inicializar punteros y dejar explícitamente claro que no se les ha reservado memoria.

vector_dinamico_1.c — **vector_dinamico.c**

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  int * a = NULL;
```

Fragmentación de la memoria

Ya hemos dicho que *malloc* puede fracasar si se solicita más memoria de la disponible en el ordenador. Parece lógico pensar que en un ordenador con 64 megabytes, de los que el sistema operativo y los programas en ejecución han consumido, digamos, 16 megabytes, podamos solicitar un bloque de hasta 48 megabytes. Pero eso no está garantizado. Imagina que los 16 megabytes ya ocupados no están dispuestos contiguamente en la memoria sino que, por ejemplo, se alternan con fragmentos de memoria libre de modo que, de cada cuatro megabytes, uno está ocupado y tres están libres, como muestra esta figura:



En tal caso, el bloque de memoria más grande que podemos obtener con *malloc* es de ¡sólo tres megabytes!

Decimos que la memoria está *fragmentada* para referirnos a la alternancia de bloques libres y ocupados que limita su disponibilidad. La fragmentación no sólo limita el máximo tamaño de bloque que puedes solicitar, además, afecta a la eficiencia con la que se ejecutan las llamadas a *malloc* y *free*.

```
    int talla, i;

    printf("Número␣de␣elementos:␣"); scanf("%d", &talla);
    a = malloc( talla * sizeof(int) );
    for (i=0; i<talla; i++)
      a[i] = i;
    free(a);
    a = NULL;

    return 0;
}
```

4.1.2. Algunos ejemplos

Es hora de poner en práctica lo aprendido desarrollando un par de ejemplos.

Creación de un nuevo vector con una selección, de talla desconocida, de elementos de otro vector

Empezaremos por diseñar una función que recibe un vector de enteros, selecciona aquellos cuyo valor es par y los devuelve en un nuevo vector cuya memoria se solicita dinámicamente.

```
int * selecciona_pares(int a[], int talla)
{
    int i, j, numpares = 0;
    int * pares;

    // Primero hemos de averiguar cuántos elementos pares hay en a.
    for (i=0; i<talla; i++)
      if (a[i] % 2 == 0)
        numpares++;

    // Ahora podemos pedir memoria para ellos.
    pares = malloc( numpares * sizeof(int) );

```

Aritmética de punteros y recorrido de vectores

La aritmética de punteros da lugar a expresiones idiomáticas de C que deberías saber leer. Fíjate en este programa:

vector_dinamico_2.c — **vector_dinamico.c**

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  int * a = NULL;
  int talla, i;
  int * p;

  printf("Número de elementos: "); scanf("%d", &talla);
  a = malloc( talla * sizeof(int) );
  for (i=0, p=a; i<talla; i++, p++)
    *p = i;
  free(a);
  a = NULL;

  return 0;
}
```

El efecto del bucle es inicializar el vector con la secuencia 0, 1, 2... El puntero *p* empieza apuntando a donde *a*, o sea, al principio del vector. Con cada autoincremento, *p*++, pasa a apuntar a la siguiente celda. Y la sentencia **p* = *i* asigna al lugar apuntado por *p* el valor *i*.

```
  // Y, finalmente, copiar los elementos pares en la zona de memoria solicitada.
  j = 0;
  for (i=0; i<talla; i++)
    if (a[i] % 2 == 0)
      pares[j++] = a[i];

  return pares;
}
```

Observa que devolvemos un dato de tipo **int ***, es decir, un puntero a entero; bueno, en realidad se trata de un puntero a una secuencia de enteros (recuerda que son conceptos equivalentes en C). Es la forma que tenemos de devolver vectores desde una función.

Este programa, por ejemplo, llama a *selecciona_pares*:

pares.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TALLA 10

  ⋮
}

int main(void)
{
  int vector[TALLA], i;
```

```
    int * seleccion;

    // Llenamos el vector con valores aleatorios.
    srand(time(0));
    for (i=0; i<TALLA; i++)
        vector[i] = rand();

    // Se efectúa ahora la selección de pares.
    seleccion = selecciona_pares(vector, TALLA);
    // La variable seleccion apunta ahora a la zona de memoria con los elementos pares.

    // Sí, pero, ¿cuántos elementos pares hay?
    for (i=0; i<????; i++)
        printf("%d\n", seleccion[i]);

    free(seleccion);
    seleccion = NULL;

    return 0;
}
```

Tenemos un problema al usar *selecciona_pares*: no sabemos cuántos valores ha seleccionado. Podemos modificar la función para que modifique el valor de un parámetro que pasamos por referencia:

```
int * selecciona_pares(int a[], int talla, int * numpares)
{
    int i, j;
    int * pares;

    // Contamos los elementos pares en el parámetro numpares, pasado por referencia.
    *numpares = 0;
    for (i=0; i<talla; i++)
        if (a[i] % 2 == 0)
            (*numpares)++;

    pares = malloc( *numpares * sizeof(int) );

    j = 0;
    for (i=0; i<talla; i++)
        if (a[i] % 2 == 0)
            pares[j++] = a[i];

    return pares;
}
```

Ahora podemos resolver el problema:

pares.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TALLA 10

int * selecciona_pares(int a[], int talla, int * numpares)
{
    int i, j;
    int * pares;
```

```

  // Contamos los elementos pares en el parámetro numpares, pasado por referencia.
  *numpares = 0;
  for (i=0; i<talla; i++)
    if (a[i] % 2 == 0)
      (*numpares)++;

  pares = malloc( *numpares * sizeof(int) );

  j = 0;
  for (i=0; i<talla; i++)
    if (a[i] % 2 == 0)
      pares[j++] = a[i];

  return pares;
}

int main(void)
{
  int vector[TALLA], i;
  int * seleccion, seleccionados;

  // Llenamos el vector con valores aleatorios.
  srand(time(0));
  for (i=0; i<TALLA; i++)
    vector[i] = rand();

  // Se efectúa ahora la selección de pares.
  seleccion = selecciona_pares(vector, TALLA, &seleccionados);
  // La variable seleccion apunta ahora a la zona de memoria con los elementos pares.
  // Además, la variable seleccionados contiene el número de pares.

  // Ahora los mostramos en pantalla.
  for (i=0; i<seleccionados; i++)
    printf("%d\n", seleccion[i]);

  free(seleccion);
  seleccion = NULL;

  return 0;
}
```

Por cierto, el prototipo de la función, que es éste:

```
int * selecciona_pares(int a[], int talla, int * seleccionados);
```

puede cambiarse por este otro:

```
int * selecciona_pares(int * a, int talla, int * seleccionados);
```

Conceptualmente, es lo mismo un parámetro declarado como **int** *a*[] que como **int** * *a*: ambos son, en realidad, punteros a enteros[3]. No obstante, es preferible utilizar la primera forma cuando un parámetro es un vector de enteros, ya que así lo distinguimos fácilmente de un entero pasado por referencia. Si ves el último prototipo, no hay nada que te permita saber si *a* es un vector o un entero pasado por referencia como *seleccionados*. Es más legible, pues, la primera forma.

[3]En realidad, hay una pequeña diferencia. La declaración **int** *a*[] hace que *a* sea un puntero inmutable, mientras que **int** * *a* permite modificar la dirección apuntada por *a* haciendo, por ejemplo, *a*++. De todos modos, no haremos uso de esa diferencia en este texto.

No puedes devolver punteros a datos locales

Como un vector de enteros y un puntero a una secuencia de enteros son, en cierto modo, equivalentes, puede que esta función te parezca correcta:

```
int * primeros(void)
{
  int i, v[10];
  for (i=0; i<10; i++)
    v[i] = i + 1;
  return v;
}
```

La función devuelve, a fin de cuentas, una dirección de memoria en la que empieza una secuencia de enteros. Y es verdad: eso es lo que hace. El problema radica en que la memoria a la que apunta ¡no «existe» fuera de la función! La memoria que ocupa *v* se libera tan pronto finaliza la ejecución de la función. Este intento de uso de la función, por ejemplo, trata de acceder ilegalmente a memoria:

```
int main(void)
{
  int * a;

  a = primeros();
  printf("%d␣", a[i]); // No existe a[i].
}
```

Recuerda: si devuelves un puntero, éste no puede apuntar a datos locales.

. . . . EJERCICIOS . . . .

▶ 219 Diseña una función que seleccione todos los números positivos de un vector de enteros. La función recibirá el vector original y un parámetro con su longitud y devolverá dos datos: un puntero al nuevo vector de enteros positivos y su longitud. El puntero se devolverá como valor de retorno de la función, y la longitud mediante un parámetro adicional (un entero pasado por referencia).

▶ 220 Desarrolla una función que seleccione todos los números de un vector de **float** mayores que un valor dado. Diseña un programa que llame correctamente a la función y muestre por pantalla el resultado.

▶ 221 Escribe un programa que lea por teclado un vector de **float** cuyo tamaño se solicitará previamente al usuario. Una vez leídos los componentes del vector, el programa copiará sus valores en otro vector distinto que ordenará con el método de la burbuja. Recuerda liberar toda memoria dinámica solicitada antes de finalizar el programa.

▶ 222 Escribe una función que lea por teclado un vector de **float** cuyo tamaño se solicitará previamente al usuario. Escribe, además, una función que reciba un vector como el leído en la función anterior y devuelva una copia suya con los mismos valores, pero ordenados de menor a mayor (usa el método de ordenación de la burbuja o cualquier otro que conozcas).

Diseña un programa que haga uso de ambas funciones. Recuerda que debes liberar toda memoria dinámica solicitada antes de finalizar la ejecución del programa.

▶ 223 Escribe una función que reciba un vector de enteros y devuelva otro con sus *n* mayores valores, siendo *n* un número menor o igual que la talla del vector original.

▶ 224 Escribe una función que reciba un vector de enteros y un valor *n*. Si *n* es menor o igual que la talla del vector, la función devolverá un vector con las *n* primeras celdas

del vector original. En caso contrario, devolverá un vector de *n* elementos con un copia del contenido del original y con valores nulos hasta completarlo.

. . . .

No resulta muy elegante que una función devuelva valores mediante **return** y, a la vez, mediante parámetros pasados por referencia. Una posibilidad es usar únicamente valores pasados por referencia:

pares_1.c — pares.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #define TALLA 10
6
7 void selecciona_pares(int a[], int talla, int * pares[], int * numpares)
8 {
9   int i, j;
10
11   *numpares = 0;
12   for (i=0; i<talla; i++)
13     if (a[i] % 2 == 0)
14       (*numpares)++;
15
16   *pares = malloc(*numpares * sizeof(int) );
17
18   j = 0;
19   for (i=0; i<talla; i++)
20     if (a[i] % 2 == 0)
21       (*pares)[j++] = a[i];
22 }
23
24 int main(void)
25 {
26   int vector[TALLA], i;
27   int * seleccion, seleccionados;
28
29   srand(time(0));
30   for (i=0; i<TALLA; i++)
31     vector[i] = rand();
32
33   selecciona_pares(vector, TALLA, &seleccion, &seleccionados);
34
35   for (i=0; i<seleccionados; i++)
36     printf("%d\n", seleccion[i]);
37
38   free(seleccion);
39   seleccion = NULL;
40
41   return 0;
42 }
```

Fíjate en la declaración del parámetro *pares* en la línea 7: es un puntero a un vector de enteros, o sea, un vector de enteros cuya dirección se suministra a la función. ¿Por qué? Porque a resultas de llamar a la función, la dirección apuntada por *pares* será una «nueva» dirección (la que obtengamos mediante una llamada a *malloc*). La línea 16 asigna un valor a **pares*. Resulta interesante que veas cómo se asigna valores al vector apuntado por **pares* en la línea 21 (los paréntesis alrededor de **pares* son obligatorios). Finalmente, observa que *seleccion* se declara en la línea 27 como un puntero a entero y que se pasa

la dirección en la que se almacena dicho puntero en la llamada a *selecciona_pares* desde la línea 33.

Hay una forma alternativa de indicar que pasamos la dirección de memoria de un puntero de enteros. La cabecera de la función *selecciona_pares* podría haberse definido así:

```
void selecciona_pares(int a[], int talla, int ** pares, int * numpares)
```

¿Ves cómo usamos un doble asterisco?

Más elegante resulta definir un registro «vector dinámico de enteros» que almacene conjuntamente tanto el vector de elementos propiamente dicho como el tamaño del vector[4]:

pares_2.c **pares.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

struct VectorDinamicoEnteros {
   int * elementos;   // Puntero a la zona de memoria con los elementos.
   int talla;         // Número de enteros almacenados en esa zona de memoria.
};

struct VectorDinamicoEnteros selecciona_pares(struct VectorDinamicoEnteros entrada)
   // Recibe un vector dinámico y devuelve otro con una selección de los elementos
   // pares del primero.
{
   int i, j;
   struct VectorDinamicoEnteros pares;

   pares.talla = 0;
   for (i=0; i<entrada.talla; i++)
      if (entrada.elementos[i] % 2 == 0)
         pares.talla++;

   pares.elementos = malloc(pares.talla * sizeof(int) );

   j = 0;
   for (i=0; i<entrada.talla; i++)
      if (entrada.elementos[i] % 2 == 0)
         pares.elementos[j++] = entrada.elementos[i];

   return pares;
}

int main(void)
{
   int i;
   struct VectorDinamicoEnteros vector, seleccionados;

   vector.talla = 10;
   vector.elementos = malloc(vector.talla * sizeof(int));
   srand(time(0));
   for (i=0; i<vector.talla; i++)
      vector.elementos[i] = rand();
```

[4]Aunque recomendemos este nuevo método para gestionar vectores de tamaño variable, has de saber, cuando menos, leer e interpretar correctamente parámetros con tipos como `int a[]`, `int *a`, `int *a[]` o `int **a`, pues muchas veces tendrás que utilizar bibliotecas escritas por otros programadores o leer código fuente de programas cuyos diseñadores optaron por estos estilos de paso de parámetros.

Valores de retorno como aviso de errores

Es habitual que aquellas funciones C que pueden dar lugar a errores nos adviertan de ellos mediante el valor de retorno. La función *malloc*, por ejemplo, devuelve el valor `NULL` cuando no consigue reservar la memoria solicitada y un valor diferente cuando sí lo consigue. La función *scanf*, que hemos estudiado como si no devolviese valor alguno, sí lo hace: devuelve el número de elementos cuyo valor ha sido efectivamente leído. Si, por ejemplo, llamamos a `scanf("%d␣%d", &a, &b)`, la función devuelve el valor 2 si todo fue bien (se leyó el contenido de dos variables). Si devuelve el valor 1, es porque sólo consiguió leer el valor de *a*, y si devuelve el valor 0, no consiguió leer ninguno de los dos. Un programa robusto debe comprobar el valor devuelto siempre que se efectúe una llamada a *scanf*; así:

```
    if ( scanf("%d␣%d", &a, &b) != 2)
      printf("Error!␣No␣conseguí␣leer␣los␣valores␣de␣a␣y␣b.\n");
    else {
      // Situación normal.
      ...
    }
```

Las rutinas que nosotros diseñamos deberían presentar un comportamiento similar. La función *selecciona_pares*, por ejemplo, podría implementarse así:

```
int selecciona_pares(int a[], int talla, int * pares[], int * numpares)
{
  int i, j;

  *numpares = 0;
  for (i=0; i<talla; i++)
    if (a[i] % 2 == 0)
      (*numpares)++;
  *pares = malloc(*numpares * sizeof(int) );
  if (*pares == NULL) { // Algo fue mal: no conseguimos la memoria.
    *numpares = 0;      // Informamos de que el vector tiene capacidad 0...
    return 0;           // y devolvemos el valor 0 para advertir de que hubo un error.
  }
  j = 0;
  for (i=0; i<talla; i++)
    if (a[i] % 2 == 0)
      (*pares)[j++] = a[i];
  return 1; // Si llegamos aquí, todo fue bien, así que avisamos de ello con el valor 1.
}
```

Aquí tienes un ejemplo de uso de la nueva función:

```
    if ( selecciona_pares(vector, TALLA, &seleccion, &seleccionados) ) {
      // Todo va bien.
    }
    else {
      // Algo fue mal.
    }
```

Hay que decir, no obstante, que esta forma de aviso de errores empieza a quedar obsoleto. Los lenguajes de programación más modernos, como C++ o Python, suelen basar la detección (y el tratamiento) de errores en las denominadas «excepciones».

```

  seleccionados = selecciona_pares(vector);

```

```
  for (i=0; i<seleccionados.talla; i++)
    printf ("%d\n", seleccionados.elementos[i]);

  free(seleccionados.elementos);
  seleccionados.elementos = NULL;
  seleccionados.talla = 0;

  return 0;
}
```

El único problema de esta aproximación es la potencial fuente de ineficiencia que supone devolver una copia de un registro, pues podría ser de gran tamaño. No es nuestro caso: un **struct** *VectorDinamicoEnteros* ocupa sólo 8 bytes. Si el tamaño fuera un problema, podríamos usar una variable de ese tipo como parámetro pasado por referencia. Usaríamos así sólo 4 bytes:

pares_3.c   pares.c

```
#include <stdio.h>
#include <stdlib.h>

struct VectorDinamicoEnteros {
  int * elementos;
  int talla;
};

void selecciona_pares(struct VectorDinamicoEnteros entrada,
                      struct VectorDinamicoEnteros * pares)
{
  int i, j;

  pares->talla = 0;
  for (i=0; i<entrada.talla; i++)
    if (entrada.elementos[i] % 2 == 0)
      pares->talla++;

  pares->elementos = malloc(pares->talla * sizeof(int) );

  j = 0;
  for (i=0; i<entrada.talla; i++)
    if (entrada.elementos[i] % 2 == 0)
      pares->elementos[j++] = entrada.elementos[i];
}

int main(void)
{
  int i;
  struct VectorDinamicoEnteros vector, seleccionados;

  vector.talla = 10;
  vector.elementos = malloc(vector.talla * sizeof(int));
  for (i=0; i<vector.talla; i++)
    vector.elementos[i] = rand();

  selecciona_pares(vector, &seleccionados);

  for (i=0; i<seleccionados.talla; i++)
    printf ("%d\n", seleccionados.elementos[i]);

  free(seleccionados.elementos);
  seleccionados.elementos = NULL;
```

```
  seleccionados.talla = 0;

  return 0;
}
```

Como ves, tienes muchas soluciones técnicamente diferentes para realizar lo mismo. Deberás elegir en función de la elegancia de cada solución y de su eficiencia.

> **Listas Python**
>
> Empieza a quedar claro que Python es un lenguaje mucho más cómodo que C para gestionar vectores dinámicos, que allí denominábamos listas. No obstante, debes tener presente que el intérprete de Python está escrito en C, así que cuando manejas listas Python estás, indirectamente, usando memoria dinámica como *malloc* y *free*.
>
> Cuando creas una lista Python con una orden como *a* = [0] * 5 o *a* = [0, 0, 0, 0, 0], estás reservando espacio en memoria para 5 elementos y asignándole a cada elemento el valor 0. La variable *a* puede verse como un simple puntero a esa zona de memoria (en realidad es algo más complejo).
>
> Cuando se pierde la referencia a una lista (por ejemplo, cambiando el valor asignado a *a*), Python se encarga de detectar automáticamente que la lista ya no es apuntada por nadie y de llamar a *free* para que la memoria que hasta ahora ocupaba pase a quedar libre.

Representación de polígonos con un número arbitrario de vértices

Desarrollemos un ejemplo más: un programa que lea los vértices de un polígono y calcule su perímetro. Empezaremos por crear un tipo de datos para almacenar los puntos de un polígono. Nuestro tipo de datos se define así:

```
struct Punto {
  float x, y;
};

struct Poligono {
  struct Punto * p;
  int puntos;
};
```

Fíjate en que un polígono presenta un número de puntos inicialmente desconocido, por lo que hemos de recurrir a memoria dinámica. Reservaremos la memoria justa para guardar dichos puntos en el campo *p* (un puntero a una secuencia de puntos) y el número de puntos se almacenará en el campo *puntos*.

Aquí tienes una función que lee un polígono por teclado y devuelve un registro con el resultado:

```
struct Poligono lee_poligono(void)
{
  int i;
  struct Poligono pol;

  printf("Número de puntos: "); scanf("%d", &pol.puntos);
  pol.p = malloc( pol.puntos * sizeof(struct Punto));
  for (i=0; i<pol.puntos; i++) {
    printf("Punto %d\n", i);
    printf("x: "); scanf("%f", &pol.p[i].x);
    printf("y: "); scanf("%f", &pol.p[i].y);
  }
  return pol;
}
```

Es interesante la forma en que solicitamos memoria para el vector de puntos:

```
pol.p = malloc( pol.puntos * sizeof(struct Punto));
```

Solicitamos memoria para *pol.puntos* celdas, cada una con capacidad para un dato de tipo **struct** *Punto* (es decir, ocupando **sizeof**(**struct** *Punto*) bytes).

Nos vendrá bien una función que libere la memoria solicitada para almacenar un polígono, ya que, de paso, pondremos el valor correcto en el campo *puntos*:

```
void libera_poligono(struct Poligono * pol)
{
  free (pol->p);
  pol->p = NULL;
  pol->puntos = 0;
}
```

Vamos ahora a definir una función que calcula el perímetro de un polígono:

```
float perimetro_poligono(struct Poligono pol)
{
  int i;
  float perim = 0.0;

  for (i=1; i<pol.puntos; i++)
    perim += sqrt( (pol.p[i].x - pol.p[i-1].x) *
                   (pol.p[i].x - pol.p[i-1].x) +
                   (pol.p[i].y - pol.p[i-1].y) *
                   (pol.p[i].y - pol.p[i-1].y) );
  perim += sqrt( (pol.p[pol.puntos-1].x - pol.p[0].x) *
                 (pol.p[pol.puntos-1].x - pol.p[0].x) +
                 (pol.p[pol.puntos-1].y - pol.p[0].y) *
                 (pol.p[pol.puntos-1].y - pol.p[0].y) );
  return perim;
}
```

Es importante que entiendas bien expresiones como *pol.p*[*i*].*x*. Esa, en particular, significa: del parámetro *pol*, que es un dato de tipo **struct** *Poligono*, accede al componente *i* del campo *p*, que es un vector de puntos; dicho componente es un dato de tipo **struct** *Punto*, pero sólo nos interesa acceder a su campo *x* (que, por cierto, es de tipo **float**).

Juntemos todas las piezas y añadamos un sencillo programa principal que invoque a las funciones desarrolladas:

poligonos_dinamicos.c   poligonos_dinamicos.c

```
#include <stdio.h>
#include <stdlib.h>

struct Punto {
  float x, y;
};

struct Poligono {
  struct Punto * p;
  int puntos;
};

struct Poligono lee_poligono(void)
{
  int i;
  struct Poligono pol;

```

```
  printf("Número de puntos: "); scanf("%d", &pol.puntos);
  pol.p = malloc( pol.puntos * sizeof(struct Punto));
  for (i=0; i<pol.puntos; i++) {
    printf("Punto %d\n", i);
    printf("x: "); scanf("%f", &pol.p[i].x);
    printf("y: "); scanf("%f", &pol.p[i].y);
  }
  return pol;
}

void libera_poligono(struct Poligono * pol)
{
  free(pol->p);
  pol->p = NULL;
  pol->puntos = 0;
}

float perimetro_poligono(struct Poligono pol)
{
  int i;
  float perim = 0.0;

  for (i=1; i<pol.puntos; i++)
    perim += sqrt( (pol.p[i].x - pol.p[i-1].x) *
                   (pol.p[i].x - pol.p[i-1].x) +
                   (pol.p[i].y - pol.p[i-1].y) *
                   (pol.p[i].y - pol.p[i-1].y) );
  perim += sqrt( (pol.p[pol.puntos-1].x - pol.p[0].x) *
                 (pol.p[pol.puntos-1].x - pol.p[0].x) +
                 (pol.p[pol.puntos-1].y - pol.p[0].y) *
                 (pol.p[pol.puntos-1].y - pol.p[0].y) );
  return perim;
}

int main(void)
{
  struct Poligono un_poligono;
  float perimetro;

  un_poligono = lee_poligono();
  perimetro = perimetro_poligono(un_poligono);
  printf("Perímetro %f\n", perimetro);
  libera_poligono(&un_poligono);

  return 0;
}
```

No es el único modo en que podríamos haber escrito el programa. Te presentamos ahora una implementación con bastantes diferencias en el modo de paso de parámetros:

poligonos_dinamicos_1.c   poligonos_dinamicos.c

```
#include <stdio.h>
#include <stdlib.h>

struct Punto {
  float x, y;
};

struct Poligono {
  struct Punto * p;
```

```
   int puntos;
};

void lee_poligono(struct Poligono * pol)
{
   int i;

   printf("Número␣de␣puntos:␣"); scanf("%d", &pol->puntos);
   pol->p = malloc( pol->puntos * sizeof(struct Punto));
   for (i=0; i<pol->puntos; i++) {
      printf("Punto␣%d\n", i);
      printf("x:␣"); scanf("%f", &pol->p[i].x);
      printf("y:␣"); scanf("%f", &pol->p[i].y);
   }
}

void libera_poligono(struct Poligono * pol)
{
   free (pol->p);
   pol->p = NULL;
   pol->puntos = 0;
}

float perimetro_poligono(const struct Poligono * pol)
{
   int i;
   float perim = 0.0;

   for (i=1; i<pol->puntos; i++)
      perim += sqrt( (pol->p[i].x - pol->p[i-1].x) *
                     (pol->p[i].x - pol->p[i-1].x) +
                     (pol->p[i].y - pol->p[i-1].y) *
                     (pol->p[i].y - pol->p[i-1].y) );
   perim += sqrt((pol->p[pol->puntos-1].x - pol->p[0].x) *
                 (pol->p[pol->puntos-1].x - pol->p[0].x) +
                 (pol->p[pol->puntos-1].y - pol->p[0].y) *
                 (pol->p[pol->puntos-1].y - pol->p[0].y) );
   return perim;
}

int main(void)
{
   struct Poligono un_poligono;
   float perimetro;

   lee_poligono(&un_poligono);
   perimetro = perimetro_poligono(&un_poligono);
   printf("Perímetro␣%f\n", perimetro);
   libera_poligono(&un_poligono);

   return 0;
}
```

En esta versión hemos optado, siempre que ha sido posible, por el paso de parámetros por referencia, es decir, por pasar la dirección de la variable en lugar de una copia de su contenido. Hay una razón para hacerlo: la eficiencia. Cada dato de tipo **struct** *Poligono* esta formado por un puntero (4 bytes) y un entero (4 bytes), así que ocupa 8 bytes. Si pasamos o devolvemos una copia de un **struct** *Poligono*, estamos copiando 8 bytes. Si, por contra, pasamos su dirección de memoria, sólo hay que pasar 4 bytes. En este

caso particular no hay una ganancia extraordinaria, pero en otras aplicaciones manejarás **struct**s tan grandes que el paso de la dirección compensará la ligera molestia de la notación de acceso a campos con el operador **->**.

Puede que te extrañe el término **const** calificando el parámetro de *perimetro_poligono*. Su uso es opcional y sirve para indicar que, aunque es posible modificar la información apuntada por *pol*, no lo haremos. En realidad suministramos el puntero por cuestión de eficiencia, no porque deseemos modificar el contenido. Con esta indicación conseguimos dos efectos: si intentásemos modificar accidentalmente el contenido, el compilador nos advertiría del error; y, si fuera posible, el compilador efectuaría optimizaciones que no podría aplicar si la información apuntada por *pol* pudiera modificarse en la función.

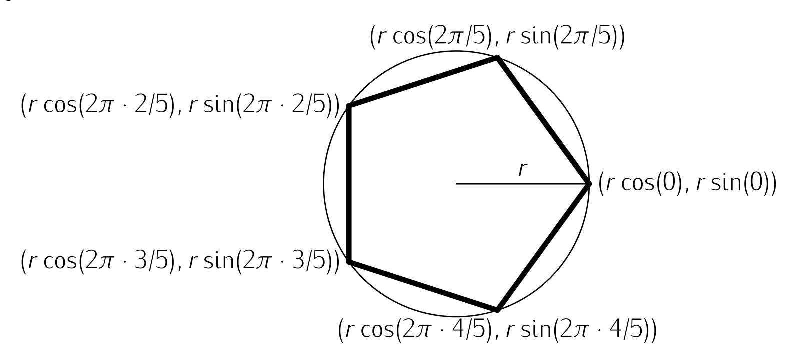
EJERCICIOS

▶ 225 ¿Funciona esta otra implementación de *perimetro_poligono*?

```
float perimetro_poligono(struct Poligono pol)
{
  int i;
  float perim = 0.0;

  for (i=1; i<pol.puntos+1; i++)
    perim += sqrt((pol.p[i%pol.puntos].x - pol.p[i-1].x) *
                  (pol.p[i%pol.puntos].x - pol.p[i-1].x)+
                  (pol.p[i%pol.puntos].y - pol.p[i-1].y) *
                  (pol.p[i%pol.puntos].y - pol.p[i-1].y));
  return perim;
}
```

▶ 226 Diseña una función que cree un polígono regular de n lados inscrito en una circunferencia de radio r. Esta figura muestra un pentágono inscrito en una circunferencia de radio r y las coordenadas de cada uno de sus vértices:



Utiliza la función para crear polígonos regulares de talla 3, 4, 5, 6, ... inscritos en una circunferencia de radio 1. Calcula a continuación el perímetro de los sucesivos polígonos y comprueba si dicho valor se aproxima a 2π.

▶ 227 Diseña un programa que permita manipular polinomios de cualquier grado. Un polinomio se representará con el siguiente tipo de registro:

```
struct Polinomio {
  float * p;
  int grado;
};
```

Como puedes ver, el campo *p* es un puntero a **float**, o sea, un vector dinámico de **float**. Diseña y utiliza funciones que hagan lo siguiente:

- Leer un polinomio por teclado. Se pedirá el grado del polinomio y, tras reservar memoria suficiente para sus coeficientes, se pedirá también el valor de cada uno de ellos.

- Evaluar un polinomio $p(x)$ para un valor dado de x.
- Sumar dos polinomios. Ten en cuenta que cada uno de ellos puede ser de diferente grado y el resultado tendrá, en principio, grado igual que el mayor grado de los operandos. (Hay excepciones; piensa cuáles.)
- Multiplicar dos polinomios.

▶ 228 Diseña un programa que solicite la talla de una serie de valores enteros y dichos valores. El programa ordenará a continuación los valores mediante el procedimiento *mergesort*. (Ten en cuenta que el vector auxiliar que necesita *merge* debe tener capacidad para el mismo número de elementos que el vector original.)

. . . .

Reserva con inicialización automática

La función *calloc* es similar a *malloc*, pero presenta un prototipo diferente y hace algo más que reservar memoria: la inicializa a cero. He aquí un prototipo (similar al) de *calloc*:

```
void * calloc(int nmemb, int size);
```

Con *calloc*, puedes pedir memoria para un vector de *talla* enteros así:

```
a = calloc(talla, sizeof(int));
```

El primer parámetro es el número de elementos y el segundo, el número de bytes que ocupa cada elemento. No hay que multiplicar una cantidad por otra, como hacíamos con *malloc*.

Todos los enteros del vector se inicializan a cero. Es como si ejecutásemos este fragmento de código:

```
a = malloc( talla * sizeof(int) );
for (i = 0; i < talla; i++) a[i] = 0;
```

¿Por qué no usar siempre *calloc*, si parece mejor que *malloc*? Por eficiencia. En ocasiones no desearás que se pierda tiempo de ejecución inicializando la memoria a cero, ya que tú mismo querrás inicializarla a otros valores inmediatamente. Recuerda que garantizar la mayor eficiencia de los programas es uno de los objetivos del lenguaje de programación C.

4.1.3. Cadenas dinámicas

Las cadenas son un caso particular de vector. Podemos usar cadenas de cualquier longitud gracias a la gestión de memoria dinámica. Este programa, por ejemplo, lee dos cadenas y construye una nueva que resulta de concatenar a éstas.

cadenas_dinamicas.c   **cadenas_dinamicas.c**

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define CAPACIDAD 80

int main(void)
{
  char cadena1[CAPACIDAD+1], cadena2[CAPACIDAD+1];
  char * cadena3;
```

```

  printf("Dame␣un␣texto:␣"); gets(cadena1);
  printf("Dame␣otro␣texto:␣"); gets(cadena2);

  cadena3 = malloc( (strlen(cadena1) + strlen(cadena2) + 1) * sizeof(char) );

  strcpy(cadena3, cadena1);
  strcat(cadena3, cadena2);

  printf("Resultado␣de␣concatenar␣ambos:␣%s\n", cadena3);

  free(cadena3);
  cadena3 = NULL;

  return 0;
}
```

Como las dos primeras cadenas se leen con *gets*, hemos de definirlas como cadenas estáticas. La tercera cadena reserva exactamente la misma cantidad de memoria que ocupa.

. . . . EJERCICIOS . . . .

▶ 229 Diseña una función que lea una cadena y construya otra con una copia invertida de la primera. La segunda cadena reservará sólo la memoria que necesite.

▶ 230 Diseña una función que lea una cadena y construya otra que contenga un ejemplar de cada carácter de la primera. Por ejemplo, si la primera cadena es `"este␣ejemplo"`, la segunda será `"est␣jmplo"`. Ten en cuenta que la segunda cadena debe ocupar la menor cantidad de memoria posible.

. . . .

Sobre la mutabilidad de las cadenas

Es posible inicializar un puntero a cadena de modo que apunte a un literal de cadena:

```
char * p = "cadena";
```

Pero, ¡ojo!, la cadena apuntada por *p* es, en ese caso, inmutable: si intentas asignar un **char** a *p*[*i*], el programa puede abortar su ejecución. ¿Por qué? Porque los literales de cadena «residen» en una zona de memoria especial (la denominada «zona de texto») que está protegida contra escritura. Y hay una razón para ello: en esa zona reside, también, el código de máquina correspondiente al programa. Que un programa modifique su propio código de máquina es una pésima práctica (que era relativamente frecuente en los tiempos en que predominaba la programación en ensamblador), hasta el punto de que su zona de memoria se marca como de sólo lectura.

. . . . EJERCICIOS . . . .

▶ 231 Implementa una función que reciba una cadena y devuelva una copia invertida. (Ten en cuenta que la talla de la cadena puede conocerse con *strlen*, así que no es necesario que suministres la talla explícitamente ni que devuelvas la talla de la memoria solicitada con un parámetro pasado por referencia.)

Escribe un programa que solicite varias palabras a un usuario y muestre el resultado de invertir cada una de ellas.

. . . .

4.2. Matrices dinámicas

Podemos extender la idea de los vectores dinámicos a matrices dinámicas. Pero el asunto se complica notablemente: no podemos gestionar la matriz como una sucesión de elementos contiguos, sino como un «vector dinámico de vectores dinámicos».

4.2.1. Gestión de memoria para matrices dinámicas

Analiza detenidamente este programa:

matriz_dinamica.c   matriz_dinamica.c

```
1  #define <stdio.h>
2  #define <stdlib.h>
3
4  int main(void)
5  {
6    float ** m = NULL;
7    int filas, columnas;
8
9    printf("Filas:␣"); scanf("%d", &filas);
10   printf("Columnas:␣"); scanf("%d", &columnas);
11
12   /* reserva de memoria */
13   m = malloc(filas * sizeof(float *));
14   for (i=0; i<filas; i++)
15     m[i] = malloc(columnas * sizeof(float));
16
17   /* trabajo con m[i][j] */
18   ...
19
20   /* liberación de memoria */
21   for (i=0; i<filas; i++)
22     free(m[i]);
23   free(m);
24   m = NULL;
25
26   return 0;
27 }
```

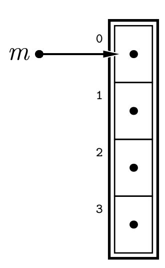
Analicemos poco a poco el programa.

Declaración del tipo

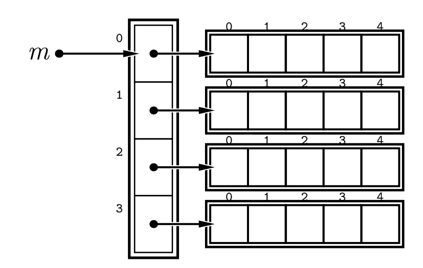
Empecemos por la declaración de la matriz (línea 6). Es un puntero un poco extraño: se declara como **float** ** *m*. Dos asteriscos, no uno. Eso es porque se trata de un *puntero a un puntero de enteros* o, equivalentemente, un *vector dinámico de vectores dinámicos de enteros*.

Reserva de memoria

Sigamos. Las líneas 9 y 10 solicitan al usuario los valores de *filas* y *columnas*. En la línea 13 encontramos una petición de memoria. Se solicita espacio para un número *filas* de punteros a **float**. Supongamos que *filas* vale 4. Tras esa petición, tenemos la siguiente asignación de memoria para *m*:



El vector *m* es un vector dinámico cuyos elementos son punteros (del tipo **float** *). De momento, esos punteros no apuntan a ninguna zona de memoria reservada. De ello se encarga la línea 15. Dicha línea está en un bucle, así que se ejecuta para *m*[0], *m*[1], *m*[2], ... El efecto es proporcionar un bloque de memoria para cada celda de *m*. He aquí el efecto final:



Acceso a filas y elementos

Bien. ¿Y cómo se usa *m* ahora? ¡Como cualquier matriz! Pensemos en qué ocurre cuando accedemos a *m*[1][2]. Analicemos *m*[1][2] de izquierda a derecha. Primero tenemos a *m*, que es un puntero (tipo **float** **), o sea, un vector dinámico a elementos del tipo **float** *. El elemento *m*[1] es el segundo componente de *m*. ¿Y de qué tipo es? De tipo **float** *, un nuevo puntero o vector dinámico, pero a valores de tipo **float**. Si es un vector dinámico, lo podemos indexar, así que es válido escribir *m*[1][2]. ¿Y de qué tipo es eso? De tipo **float**. Fíjate:

- *m* es de tipo **float** **;
- *m*[1] es de tipo **float** *;
- *m*[1][2] es de tipo **float**.

Con cada indexación, «desaparece» un asterisco del tipo de datos.

Liberación de memoria: un *free* para cada *malloc*

Sigamos con el programa. Nos resta la liberación de memoria. Observa que hay una llamada a *free* por cada llamada a *malloc* realizada con anterioridad (líneas 20–24). Hemos de liberar cada uno de los bloques reservados y hemos de empezar a hacerlo por los de «segundo nivel», es decir, por los de la forma *m*[*i*]. Si empezásemos liberando *m*, cometeríamos un grave error: si liberamos *m* antes que todos los *m*[*i*], perderemos el puntero que los referencia y, en consecuencia, ¡no podremos liberarlos!

```
  ...
  free(m);
  m = NULL;
  /* liberación de memoria incorrecta: ¿qué es m[i] ahora que m vale NULL? */
  for (i=0; i<filas; i++)
    free(m[i]);
}
```

Más eficiencia, menos reservas de memoria

Te hemos enseñado una forma «estándar» de pedir memoria para matrices dinámicas. No es la única. Es más, no es la más utilizada en la práctica. ¿Por qué? Porque obliga a realizar tantas llamadas a *malloc* (y después a *free*) como filas tiene la matriz más uno. Las llamadas a *malloc* pueden resultar ineficientes cuando su número es grande. Es posible reservar la memoria de una matriz dinámica con sólo dos llamadas a *malloc*.

```
#include <stdlib.h>

int main(void)
{
  int ** m;
  int filas, columnas;

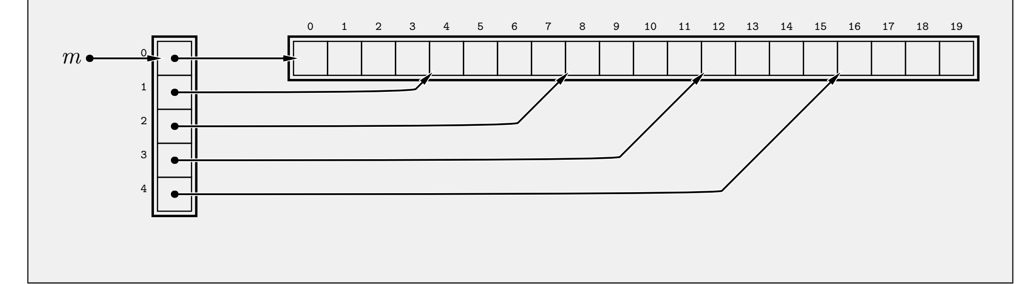
  filas = ...;
  columnas = ...;

  // Reserva de memoria.
  m = malloc(filas * sizeof(int *));
  m[0] = malloc(filas * columnas * sizeof(int));
  for (i=1; i<filas; i++) m[i] = m[i-1] + columnas;

  ...
  // Liberación de memoria.
  free(m[0]);
  free(m);

  return 0;
}
```

La clave está en la sentencia m[*i*] = *m*[*i*-1] + *columnas*: el contenido de *m*[*i*] pasa a ser la dirección de memoria *columnas* celdas más a la derecha de la dirección *m*[*i*-1]. He aquí una representación gráfica de una matriz de 5×4:



Matrices dinámicas y funciones

El paso de matrices dinámicas a funciones tiene varias formas idiomáticas que conviene que conozcas. Imagina una función que recibe una matriz de enteros para mostrar su contenido por pantalla. En principio, la cabecera de la función presentaría este aspecto:

```
void muestra_matriz(int ** m)
```

El parámetro indica que es de tipo «puntero a punteros a enteros». Una forma alternativa de decir lo mismo es ésta:

```
void muestra_matriz(int * m[])
```

Se lee más bien como «vector de punteros a entero». Pero ambas expresiones son sinónimas de «vector de vectores a entero». Uno se siente tentado de utilizar esta otra cabecera:

```
void muestra_matriz(int m[][]) // ¡Mal!
```

Pero no funciona. Es incorrecta. C entiende que queremos pasar una matriz estática y que hemos omitido el número de columnas.

Sigamos con la función:

```
void muestra_matriz(int ** m )
{
   int i,j;

   for (i=0; i<???; i++) {
      for (j=0; j<???; j++)
         printf("%d␣", m[i][j]);
      printf("\n");
   }
}
```

Observa que necesitamos suministrar el número de filas y columnas explícitamente para saber qué rango de valores deben tomar *i* y *j*:

```
void muestra_matriz(int ** m, int filas, int columnas)
{
   int i,j;

   for (i=0; i<filas; i++) {
      for (j=0; j<columnas; j++)
         printf("%d␣", m[i][j]);
      printf("\n");
   }
}
```

Supongamos ahora que nos piden una función que efectúe la liberación de la memoria de una matriz:

```
void libera_matriz(int ** m, int filas, int columnas)
{
   int i;

   for (i=0; i<filas; i++)
      free(m[i]);
   free(m);
}
```

Ahora resulta innecesario el paso del número de columnas, pues no se usa en la función:

```
void libera_matriz(int ** m, int filas)
{
   int i;

   for (i=0; i<filas; i++)
      free(m[i]);
   free(m);
}
```

Falta un detalle que haría mejor a esta función: la asignación del valor `NULL` a *m* al final de todo. Para ello tenemos que pasar una referencia a la matriz, y no la propia matriz:

```
void libera_matriz(int *** m, int filas)
{
   int i;

```

```
  for (i=0; i<filas; i++)
    free( (*m)[i]);
  free(*m);
  *m = NULL;
}
```

¡Qué horror! ¡Tres asteriscos en la declaración del parámetro *m*! C no es, precisamente, el colmo de la elegancia.

EJERCICIOS

▶ 232 Diseña una *función* que reciba un número de filas y un número de columnas y devuelva una matriz dinámica de enteros con *filas*×*columnas* elementos.

▶ 233 Diseña un *procedimiento* que reciba un puntero a una matriz dinámica (sin memoria asignada), un número de filas y un número de columnas y devuelva, mediante el primer parámetro, una matriz dinámica de enteros con *filas*×*columnas* elementos.

La gestión de matrices dinámicas considerando por separado sus tres variables (puntero a memoria, número de filas y número de columnas) resulta poco elegante y da lugar a funciones con parámetros de difícil lectura. En el siguiente apartado aprenderás a usar matrices dinámicas que agrupan sus tres datos en un tipo registro definido por el usuario.

4.2.2. Definición de un tipo «matriz dinámica» y de funciones para su gestión

Presentaremos ahora un ejemplo de aplicación de lo aprendido: un programa que multiplica dos matrices de tallas arbitrarias. Empezaremos por definir un nuevo tipo de datos para nuestras matrices. El nuevo tipo será un **struct** que contendrá una matriz dinámica de **float** y el número de filas y columnas.

```
struct Matriz {
  float ** m;
  int filas, columnas;
};
```

Diseñemos ahora una función que «cree» una matriz dado el número de filas y el número de columnas:

```
struct Matriz crea_matriz (int filas, int columnas)
{
  struct Matriz mat;
  int i;

  if (filas <= 0 || columnas <=0) {
    mat.filas = mat.columnas = 0;
    mat.m = NULL;
    return mat;
  }

  mat.filas = filas;
  mat.columnas = columnas;
  mat.m = malloc ( filas * sizeof(float *) );
  for (i=0; i<filas; i++)
    mat.m[i] = malloc ( columnas * sizeof(float) );
  return mat;
}
```

Hemos tenido la precaución de no pedir memoria si el número de filas o columnas no son válidos. Para crear una matriz de, por ejemplo, 3×4, llamaremos a la función así:

```
struct Matriz matriz;
...
matriz = crea_matriz(3, 4);
```

Hay una implementación alternativa de *crea_matriz*:

```
void crea_matriz (int filas, int columnas, struct Matriz * mat)
{
  int i;

  if (filas <= 0 || columnas <=0) {
    mat->filas = mat->columnas = 0;
    mat->m = NULL;
  }
  else {
    mat->filas = filas;
    mat->columnas = columnas;
    mat->m = malloc ( filas * sizeof(float *) );
    for (i=0; i<filas; i++)
      mat->m[i] = malloc ( columnas * sizeof(float) );
  }
}
```

En este caso, la función (procedimiento) se llamaría así:

```
struct Matriz matriz;
...
crea_matriz(3, 4, &matriz);
```

También nos vendrá bien disponer de un procedimiento para liberar la memoria de una matriz:

```
void libera_matriz (struct Matriz * mat)
{
  int i;

  if (mat->m != NULL) {
    for (i=0; i<mat->filas; i++)
      free(mat->m[i]);
    free(mat->m);
  }

  mat->m = NULL;
  mat->filas = 0;
  mat->columnas = 0;
}
```

Para liberar la memoria de una matriz dinámica *m*, efectuaremos una llamada como ésta:

```
libera_matriz(&m);
```

Como hemos de leer dos matrices por teclado, diseñemos ahora una función capaz de leer una matriz por teclado:

```
struct Matriz lee_matriz (void)
{
  int i, j, filas, columnas;
  struct Matriz mat;

  printf("Filas:␣"); scanf("%d", &filas);
```

```
  printf ("Columnas:␣"); scanf ("%d", &columnas);

  mat = crea_matriz(filas, columnas);

  for (i=0; i<filas; i++)
    for (j=0; j<columnas; j++) {
      printf ("Elemento␣[%d][%d]:␣", i, j); scanf ("%f", &mat.m[i][j]);
    }
  return mat;
}
```

Observa que hemos llamado a *crea_matriz* tan pronto hemos sabido cuál era el número de filas y columnas de la matriz.

Y ahora, implementemos un procedimiento que muestre por pantalla una matriz:

```
void muestra_matriz (struct Matriz mat)
{
  int i, j;

  for (i=0; i<mat.filas; i++) {
    for (j=0; j<mat.columnas; j++)
      printf ("%f␣", mat.m[i][j]);
    printf ("\n");
  }
}
```

. . . . EJERCICIOS . . . .

▶ 234 En *muestra_matriz* hemos pasado la matriz *mat* por valor. ¿Cuántos bytes se copiarán en pila con cada llamada?

▶ 235 Diseña una nueva versión de *muestra_matriz* en la que *mat* se pase por referencia. ¿Cuántos bytes se copiarán en pila con cada llamada?

. . . .

Podemos proceder ya mismo a implementar una función que multiplique dos matrices:

```
struct Matriz multiplica_matrices (struct Matriz a, struct Matriz b)
{
  int i, j, k;
  struct Matriz c;

  if (a.columnas != b.filas) { /* No se pueden multiplicar */
    c.filas = c.columnas = 0;
    c.m = NULL;
    return c;
  }
  c = crea_matriz(a.filas, b.columnas);
  for (i=0; i<c.filas; i++)
    for (j=0; j<c.columnas; j++) {
      c.m[i][j] = 0.0;
      for (k=0; k<a.columnas; k++)
        c.m[i][j] += a.m[i][k] * b.m[k][j];
    }
  return c;
}
```

No todo par de matrices puede multiplicarse entre sí. El número de columnas de la primera ha de ser igual al número de filas de la segunda. Por eso devolvemos una matriz vacía (de 0×0) cuando *a.columnas* es distinto de *b.filas*.

Ya podemos construir el programa principal:

```c
#include <stdio.h>

...definición de funciones...

int main(void)
{
  struct Matriz a, b, c;

  a = lee_matriz();
  b = lee_matriz();
  c = multiplica_matrices(a, b);
  if (c.m == NULL)
    printf("Las matrices no son multiplicables\n");
  else {
    printf("Resultado del producto:\n");
    muestra_matriz(c);
  }
  libera_matriz(&a);
  libera_matriz(&b);
  libera_matriz(&c);

  return 0;
}
```

EJERCICIOS

▶ 236 Diseña una función que sume dos matrices.

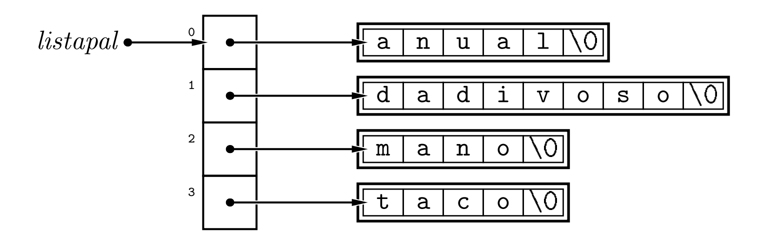
▶ 237 Pasar estructuras por valor puede ser ineficiente, pues se debe obtener una copia en pila de la estructura completa (en el caso de las matrices, cada variable de tipo **struct** *Matriz* ocupa 12 bytes —un puntero y dos enteros—, cuando una referencia supone la copia de sólo 4 bytes). Modifica la función que multiplica dos matrices para que sus dos parámetros se pasen por referencia.

▶ 238 Diseña una función que encuentre, si lo hay, un *punto de silla* en una matriz. Un punto de silla es un elemento de la matriz que es o bien el máximo de su fila y el mínimo de su columna a la vez, o bien el mínimo de su fila y el máximo de su columna a la vez. La función devolverá cierto o falso dependiendo de si hay algún punto de silla. Si lo hay, el valor del primer punto de silla encontrado se devolverá como valor de un parámetro pasado por referencia.

4.3. Más allá de las matrices dinámicas

4.3.1. Vectores de vectores de tallas arbitrarias

Hemos aprendido a definir matrices dinámicas con un vector dinámico de vectores dinámicos. El primero contiene punteros que apuntan a cada columna. Una característica de las matrices es que todas las filas tienen el mismo número de elementos (el número de columnas). Hay estructuras similares a las matrices pero que no imponen esa restricción. Pensemos, por ejemplo, en una lista de palabras. Una forma de almacenarla en memoria es la que se muestra en este gráfico:



¿Ves? Es parecido a una matriz, pero no *exactamente* una matriz: cada palabra ocupa tanta memoria como necesita, pero no más. Este programa solicita al usuario 4 palabras y las almacena en una estructura como la dibujada:

cuatro_palabras.c — cuatro_palabras.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PALS 4
#define MAXLON 80

int main(void)
{
  char ** listapal;
  char linea[MAXLON+1];
  int i;

  /* Pedir memoria y leer datos */
  listapal = malloc(PALS * sizeof(char *));
  for (i=0; i<PALS; i++) {
    printf("Teclea␣una␣palabra:␣");
    gets(linea);
    listapal[i] = malloc( (strlen(linea)+1) * sizeof(char) );
    strcpy(listapal[i], linea);
  }

  /* Mostrar el contenido de la lista */
  for (i=0; i<PALS; i++)
    printf("Palabra␣%i:␣%s\n", i, listapal[i]);

  /* Liberar memoria */
  for (i=0; i<PALS; i++)
    free(listapal[i]);
  free(listapal);

  return 0;
}
```

Este otro programa sólo usa memoria dinámica para las palabras, pero no para el vector de palabras:

cuatro_palabras_1.c — cuatro_palabras.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define PALS 4
#define MAXLON 80

int main(void)
{
```

```
    char * listapal[PALS];
    char linea[MAXLON+1];
    int i;

    /* Pedir memoria y leer datos */
    for (i=0; i<PALS; i++) {
      printf("Teclea una palabra: ");
      gets(linea);
      listapal[i] = malloc( (strlen(linea)+1) * sizeof(char) );
      strcpy(listapal[i], linea);
    }

    /* Mostrar el contenido de la lista */
    for (i=0; i<PALS; i++)
      printf("Palabra %i: %s\n", i, listapal[i]);

    /* Liberar memoria */
    for (i=0; i<PALS; i++)
      free(listapal[i]);

    return 0;
}
```

Fíjate en cómo hemos definido *listapal*: como un vector estático de 4 punteros a caracteres (**char** * *listapal*[PALS]).

Vamos a ilustrar el uso de este tipo de estructuras de datos con la escritura de una función que reciba una cadena y devuelva un vector de palabras, es decir, vamos a implementar la funcionalidad que ofrece Python con el método *split*. Empecemos por considerar la cabecera de la función, a la que llamaremos *extrae_palabras*. Está claro que uno de los parámetros de entrada es una cadena, o sea, un vector de caracteres:

```
??? extrae_palabras(char frase[], ???)
```

No hace falta suministrar la longitud de la cadena, pues ésta se puede calcular con la función *strlen*. ¿Cómo representamos la información de salida? Una posibilidad es devolver un vector de cadenas:

```
char ** extrae_palabras(char frase[], ???)
```

O sea, devolvemos un puntero (*) a una serie de datos de tipo **char** *, o sea, cadenas. Pero aún falta algo: hemos de indicar explícitamente cuántas palabras hemos encontrado:

```
char ** extrae_palabras(char frase[], int * numpals)
```

Hemos recurrido a un parámetro adicional para devolver el segundo valor. Dicho parámetro es la dirección de un entero, pues vamos a modificar su valor. Ya podemos codificar el cuerpo de la función. Empezaremos por contar las palabras, que serán series de caracteres separadas por blancos (no entraremos en mayores complicaciones acerca de qué es una palabra).

```
char ** extrae_palabras(char frase[], int * numpals)
{
  int i, lonfrase;

  lonfrase = strlen(frase);
  *numpals = 1;
  for (i=0; i<lonfrase-1; i++)
    if (frase[i] == ' ' && frase[i+1] != ' ') (*numpals)++;
  if (frase[0] == ' ') (*numpals)--;

  ...
}
```

Acceso a argumentos de la línea de comandos

Los programas que diseñamos en el curso suponen que *main* no tiene parámetros. No siempre es así.

La función *main* puede recibir como argumentos las opciones que se indican en la línea de comandos cuando ejecutas el programa desde la línea de órdenes Unix. El siguiente programa muestra por pantalla un saludo personalizado y debe llamarse así:

```
saluda -n nombre
```

Aquí tienes el código fuente:

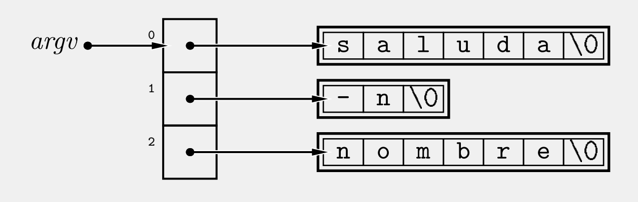
saluda.c

```c
#include <stdio.h>
#include <string.h>

main (int argc, char * argv[])
{
  if (argc != 3)
    printf ("Error:␣necesito␣que␣indiques␣el␣nombre␣con␣-n\n");
  else
    if (strcmp(argv[1], "-n") != 0)
      printf ("Error:␣sólo␣entiendo␣la␣opción␣-n\n");
    else
      printf ("Hola,␣%s.", argv[2]);
}
```

El argumento *argc* indica cuántas «palabras» se han usado en la línea de órdenes. El argumento *argv* es un vector de **char** *, es decir, un vector de cadenas (una cadena es un vector de caracteres). El elemento *argv*[0] contiene el nombre del programa (en nuestro caso, "saluda") que es la primera «palabra», *argv*[1] el de la segunda (que esperamos que sea "-n") y *argv*[2] la tercera (el nombre de la persona a la que saludamos).

La estructura *argv*, tras la invocación `saluda -n nombre`, es:



Ya podemos reservar memoria para el vector de cadenas, pero aún no para cada una de ellas:

```c
char ** extrae_palabras(char frase[], int * numpals)
{
  int i, lonfrase;
  char **palabras;

  lonfrase = strlen(frase);
  *numpals = 1;
  for (i=0; i<lonfrase-1; i++)
    if (frase[i] == '␣' && frase[i+1] != '␣') (*numpals)++;
  if (frase[0] == '␣') (*numpals)--;

  palabras = malloc(*numpals * sizeof(char *));

```

```
  ...
}
```

Ahora pasamos a reservar memoria para cada una de las palabras y, tan pronto hagamos cada reserva, «escribirla» en su porción de memoria:

```
char ** extrae_palabras(char frase[], int * numpals)
{
  int i, j, inicio_pal, longitud_pal, palabra_actual, lonfrase;
  char **palabras;

  lonfrase = strlen(frase);
  *numpals = 1;
  for (i=0; i<lonfrase-1; i++)
    if (frase[i] == '␣' && frase[i+1] != '␣')
      (*numpals)++;
  if (frase[0] == '␣')
    (*numpals)--;

  palabras = malloc(*numpals * sizeof(char *));

  palabra_actual = 0;
  i = 0;
  if (frase[0] == '␣')
    while (frase[++i] == '␣' && i < lonfrase); // Saltamos blancos iniciales.

  while (i<lonfrase) {
    inicio_pal = i;
    while (frase[++i] != '␣' && i < lonfrase); // Recorremos la palabra.

    longitud_pal = i - inicio_pal; // Calculamos número de caracteres en la palabra actual.
    // Reservamos memoria.
    palabras[palabra_actual] = malloc((longitud_pal+1)*sizeof(char));

    // Y copiamos la palabra de frase al vector de palabras.
    for (j=inicio_pal; j<i; j++)
      palabras[palabra_actual][j-inicio_pal] = frase[j];
    palabras[palabra_actual][j-inicio_pal] = '\0';

    while (frase[++i] == '␣' && i < lonfrase); // Saltamos blancos entre palabras.

    palabra_actual++; // Y pasamos a la siguiente.
  }

  return palabras;
}
```

¡Buf! Complicado, ¿verdad? Veamos cómo se puede usar la función desde el programa principal:

palabras.c — palabras.c

```
#include <stdio.h>
#include <stdlib.h>

⋮
}

int main(void)
{
    char linea[MAXLON+1];
```

```c
    int numero_palabras, i;
    char ** las_palabras;

    printf("Escribe una frase: ");
    gets(linea);
    las_palabras = extrae_palabras(linea, &numero_palabras);
    for (i=0; i<numero_palabras; i++)
      printf("%s\n", las_palabras[i]);

    return 0;
}
```

¿Ya? Aún no. Aunque este programa compila correctamente y se ejecuta sin problemas, hemos de considerarlo incorrecto: hemos solicitado memoria con *malloc* y no la hemos liberado con *free*.

palabras_1.c **palabras.c**

```c
#include <stdio.h>
#include <stdlib.h>

⋮
}

int main(void)
{
    char linea[MAXLON+1];
    int numero_palabras, i;
    char ** las_palabras;

    printf("Escribe una frase: ");
    gets(linea);
    las_palabras = extrae_palabras(linea, &numero_palabras);
    for (i=0; i<numero_palabras; i++)
      printf("%s\n", las_palabras[i]);

    for (i=0; i<numero_palabras; i++)
      free(las_palabras[i]);
    free(las_palabras);
    las_palabras = NULL;

    return 0;
}
```

Ahora sí.

4.3.2. Arreglos *n*-dimensionales

Hemos considerado la creación de estructuras bidimensionales (matrices o vectores de vectores), pero nada impide definir estructuras con más dimensiones. Este sencillo programa pretende ilustrar la idea creando una estructura dinámica con $3 \times 3 \times 3$ elementos, inicializarla, mostrar su contenido y liberar la memoria ocupada:

tridimensional.c **tridimensional.c**

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int *** a; // Tres asteriscos: vector de vectores de vectores de enteros.
```

```
  int i, j, k;

  // Reserva de memoria
  a = malloc(3*sizeof(int **));
  for (i=0; i<3; i++) {
    a[i] = malloc(3*sizeof(int *));
    for (j=0; j<3; j++)
      a[i][j] = malloc(3*sizeof(int));
  }

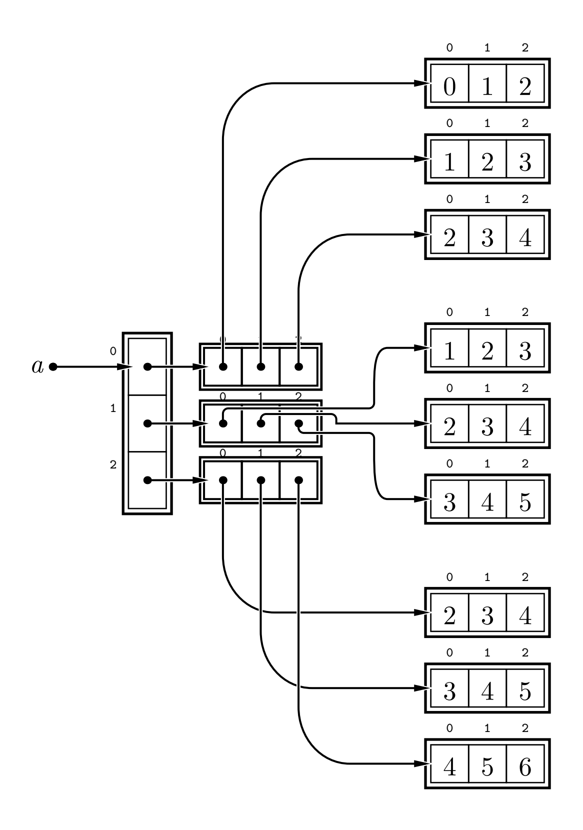
  // Inicialización
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      for (k=0; k<3; k++)
        a[i][j][k] = i+j+k;

  // Impresión
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      for (k=0; k<3; k++)
        printf("%d %d %d: %d\n", i, j, k, a[i][j][k]);

  // Liberación de memoria.
  for (i=0; i<3; i++) {
    for (j=0; j<3; j++)
      free(a[i][j]);
    free(a[i]);
  }
  free(a);
  a = NULL;

  return 0;
}
```

En la siguiente figura se muestra el estado de la memoria tras la inicialización de la matriz tridimensional:



4.4. Redimensionamiento de la reserva de memoria

Muchos programas no pueden determinar el tamaño de sus vectores antes de empezar a trabajar con ellos. Por ejemplo, cuando se inicia la ejecución de un programa que gestiona una agenda telefónica no sabemos cuántas entradas contendrá finalmente. Podemos fijar un número máximo de entradas y pedir memoria para ellas con *malloc*, pero entonces estaremos reproduciendo el problema que nos llevó a presentar los vectores dinámicos. Afortunadamente, C permite que el tamaño de un vector cuya memoria se ha solicitado previamente con *malloc* crezca en función de las necesidades. Se usa para ello la función *realloc*, cuyo prototipo es (similar a) éste:

stdlib.h

```
...
void * realloc(void * puntero, int bytes);
...
```

Aquí tienes un ejemplo de uso:

```
#include <stdlib.h>

int main(void)
{
  int * a;

  a = malloc(10 * sizeof(int));    // Se pide espacio para 10 enteros.
  ...
  a = realloc(a, 20 * sizeof(int)); // Ahora se amplía para que quepan 20.
  ...
  a = realloc(a, 5 * sizeof(int));  // Y ahora se reduce a sólo 5 (los 5 primeros).
  ...
  free(a);

  return 0;
}
```

La función *realloc* recibe como primer argumento el puntero que indica la zona de memoria que deseamos redimensionar y como segundo argumento, el número de bytes que deseamos asignar ahora. La función devuelve el puntero a la nueva zona de memoria.

¿Qué hace exactamente *realloc*? Depende de si se pide *más* o *menos* memoria de la que ya se tiene reservada:

- Si se pide más memoria, intenta primero ampliar la zona de memoria asignada. Si las posiciones de memoria que siguen a las que ocupa *a* en ese instante están libres, se las asigna a *a*, sin más. En caso contrario, solicita una nueva zona de memoria, copia el contenido de la vieja zona de memoria en la nueva, libera la vieja zona de memoria y nos devuelve el puntero a la nueva.

- Si se pide menos memoria, libera la que sobra en el bloque reservado. Un caso extremo consiste en llamar a *realloc* con una petición de 0 bytes. En tal caso, la llamada a *realloc* es equivalente a *free*.

Al igual que *malloc*, si *realloc* no puede atender una petición, devuelve un puntero a `NULL`. Una cosa más: si se llama a *realloc* con el valor `NULL` como primer parámetro, *realloc* se comporta como si se llamara a *malloc*.

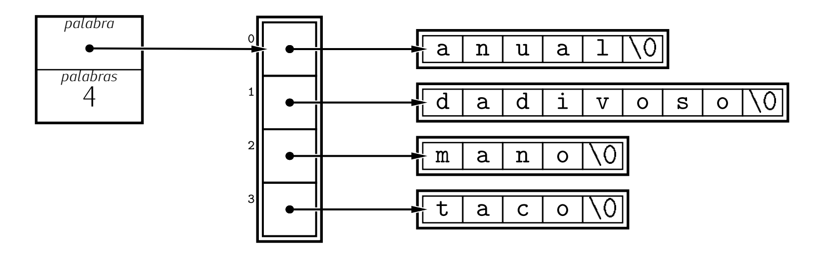
Como puedes imaginar, un uso constante de *realloc* puede ser una fuente de ineficiencia. Si tienes un vector que ocupa un 1 megabyte y usas *realloc* para que ocupe 1.1 megabyes, es probable que provoques una copia de 1 megabyte de datos de la zona vieja a la nueva. Es más, puede que ni siquiera tengas memoria suficiente para efectuar la nueva reserva, pues durante un instante (mientras se efectúa la copia) estarás usando 2.1 megabytes.

Desarrollemos un ejemplo para ilustrar el concepto de reasignación o redimensionamiento de memoria. Vamos a diseñar un módulo que permita crear diccionarios de palabras. Un diccionario es un vector de cadenas. Cuando creamos el diccionario, no sabemos cuántas palabras albergará ni qué longitud tiene cada una de las palabras. Tendremos que usar, pues, memoria dinámica. Las palabras, una vez se introducen en el diccionario, no cambian de tamaño, así que bastará con usar *malloc* para gestionar sus reservas de memoria. Sin embargo, la talla de la lista de palabras sí varía al añadir palabras, así que deberemos gestionarla con *malloc*/*realloc*.

Empecemos por definir el tipo de datos para un diccionario.

```
struct Diccionario {
   char ** palabra;
   int palabras;
};
```

Aquí tienes un ejemplo de diccionario que contiene 4 palabras:



Un diccionario vacío no tiene palabra alguna ni memoria reservada. Esta función crea un diccionario:

```
struct Diccionario crea_diccionario(void)
{
   struct Diccionario d;
   d.palabra = NULL;
```

```
    d.palabras = 0;
    return d;
}
```

Ya podemos desarrollar la función que inserta una palabra en el diccionario. Lo primero que hará la función es comprobar si la palabra ya está en el diccionario. En tal caso, no hará nada:

```
void inserta_palabra_en_diccionario(struct Diccionario * d, char pal[])
{
    int i;

    for (i=0; i<d->palabras; i++)
        if (strcmp(d->palabra[i], pal)==0)
            return;
    ...
}
```

Si la palabra no está, hemos de añadirla:

```
void inserta_palabra_en_diccionario(struct Diccionario * d, char pal[])
{
    int i;

    for (i=0; i<d->palabras; i++)
        if (strcmp(d->palabra[i], pal)==0)
            return;

    d->palabra = realloc(d->palabra, (d->palabras+1) * sizeof(char *));
    d->palabra[d->palabras] = malloc((strlen(pal)+1) * sizeof(char));
    strcpy(d->palabra[d->palabras], pal);
    d->palabras++;
}
```

Podemos liberar la memoria ocupada por un diccionario cuando no lo necesitamos más llamando a esta otra función:

```
void libera_diccionario(struct Diccionario * d)
{
    int i;

    if (d->palabra != NULL) {
        for (i=0; i<d->palabras; i++)
            free(d->palabra[i]);
        free(d->palabra);
        d->palabra = NULL;
        d->palabras = 0;
    }
}
```

EJERCICIOS

▶ 239 Diseña una función que devuelva cierto (valor 1) o falso (valor 0) en función de si una palabra pertenece o no a un diccionario.

▶ 240 Diseña una función que borre una palabra del diccionario.

▶ 241 Diseña una función que muestre por pantalla todas la palabras del diccionario que empiezan por un prefijo dado (una cadena).

No es lo mismo un puntero que un vector

Aunque C permite considerarlos una misma cosa en muchos contextos, hay algunas diferencias entre un puntero a una serie de enteros, por ejemplo, y un vector de enteros. Consideremos un programa con estas declaraciones:

```
int vector[10];
int escalar;
int * puntero;
int * otro_puntero;
```

- A los punteros debe asignárseles explícitamente algún valor:
  - a la «nada»: *puntero* = NULL;
  - a memoria reservada mediante *malloc*: *puntero* = *malloc*(5*sizeof(int));
  - a la dirección de memoria de una variable escalar del tipo al que puede apuntar el puntero: *puntero* = &*escalar*;
  - a la dirección de memoria en la que empieza un vector: *puntero* = *vector*;
  - a la dirección de memoria de un elemento de un vector: *puntero* = &*vector*[2];
  - a la dirección de memoria apuntada por otro puntero: *puntero* = *otro_puntero*;
  - a una dirección calculada mediante aritmética de punteros: por ejemplo, *puntero* = *vector*+2 es lo mismo que *puntero* = &*vector*[2].
- Los vectores reservan memoria automáticamente, pero no puedes redimensionarlos. Es ilegal, por ejemplo, una sentencia como ésta: *vector* = *puntero*.

Eso sí, las funciones que admiten el paso de un vector vía parámetros, admiten también un puntero y viceversa. De ahí que se consideren equivalentes.

Aunque suponga una simplificación, puedes considerar que un vector es un puntero inmutable (de valor constante).

▶ 242 Diseña una función que muestre por pantalla todas la palabras del diccionario que acaban con un sufijo dado (una cadena).

. . . .

La función que determina si una palabra pertenece o no a un diccionario requiere tanto más tiempo cuanto mayor es el número de palabras del diccionario. Es así porque el diccionario está desordenado y, por tanto, la única forma de estar seguros de que una palabra no está en el diccionario es recorrer todas y cada una de las palabras (si, por contra, la palabra está en el diccionario, no *siempre* es necesario recorrer el listado completo).

Podemos mejorar el comportamiento de la rutina de búsqueda si mantenemos el diccionario siempre ordenado. Para ello hemos de modificar la función de inserción de palabras en el diccionario:

```
void inserta_palabra_en_diccionario(struct Diccionario *d, char pal[])
{
  int i, j;

  for (i=0; i<d->palabras; i++) {
    if (strcmp(d->palabra[i], pal)==0) // Si ya está, no hay nada que hacer.
      return;
    if (strcmp(d->palabra[i], pal)>0)  // Encontramos su posición (orden alfabético)
      break;
```

```
10     }
11
12     /* Si llegamos aquí, la palabra no está y hay que insertarla en la posición i, desplazando
13          antes el resto de palabras. */
15
16     /* Reservamos espacio en la lista de palabras para una más. */
17     d->palabra = realloc(d->palabra, (d->palabras+1) * sizeof(char *));
18     /* Desplazamos el resto de palabras para que haya un hueco en el vector. */
19     for (j=d->palabras; j>i; j--) {
20       d->palabra[j] = malloc(strlen(d->palabra[j-1])+1)*sizeof(char));
21       strcpy(d->palabra[j], d->palabra[j-1]);
22       free(d->palabra[j-1]);
23     }
24     /* Y copiamos en su celda la nueva palabra */
25     d->palabra[i] = malloc( (strlen(pal)+1) * sizeof(char) );
26     strcpy(d->palabra[i], pal);
27     d->palabras++;
28 }
```

¡Buf! Las líneas 20–22 no hacen más que asignar a una palabra el contenido de otra (la que ocupa la posición *j* recibe una copia del contenido de la que ocupa la posición *j*-1). ¿No hay una forma mejor de hacer eso mismo? Sí. Transcribimos nuevamente las últimas líneas del programa, pero con una sola sentencia que sustituye a las líneas 20–22:

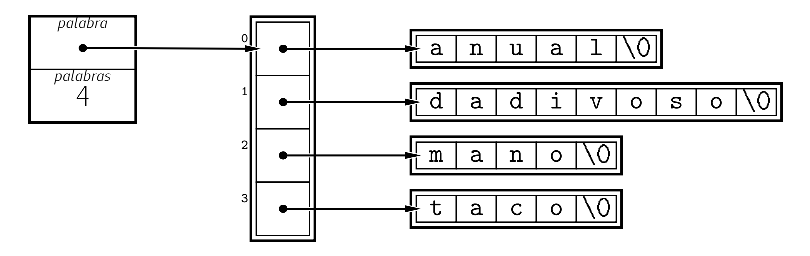
```
18     ...
19     for (j=d->palabras; j>i; i--)
20       d->palabra[j] = d->palabra[j-1];
21     /* Y copiamos en su celda la nueva palabra */
22     d->palabra[i] = malloc( (strlen(pal)+1) * sizeof(char) );
23     strcpy(d->palabra[i], pal);
24     d->palabras++;
25 }
```

No está mal, pero ¡no hemos pedido ni liberado memoria dinámica! ¡Ni siquiera hemos usado *strcpy*, y eso que dijimos que había que usar esa función para asignar una cadena a otra. ¿Cómo es posible? Antes hemos de comentar qué significa una asignación como ésta:

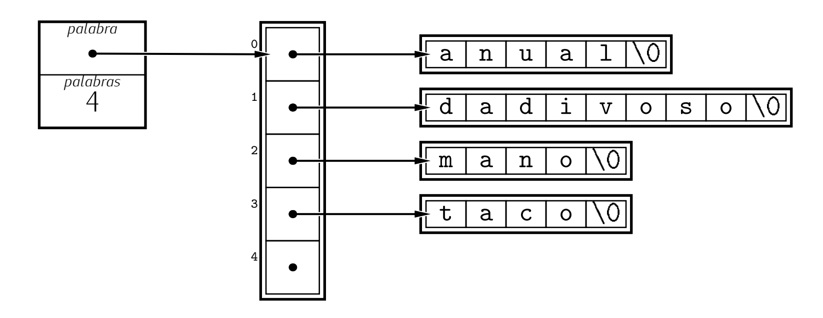
```
d->palabra[j] = d->palabra[j-1];
```

Significa que *d->palabra*[*j*] apunta al mismo lugar al que apunta *d->palabra*[*j*-1]. ¿Por qué? Porque un puntero no es más que una dirección de memoria y asignar a un puntero el valor de otro hace que ambos contengan la misma dirección de memoria, es decir, que ambos apunten al mismo lugar.

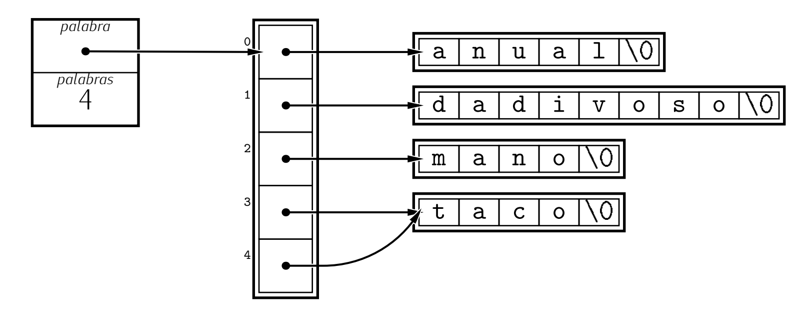
Veamos qué pasa estudiando un ejemplo. Imagina un diccionario en el que ya hemos insertado las palabras «anual», «dadivoso», «mano» y «taco» y que vamos a insertar ahora la palabra «feliz». Partimos, pues, de esta situación:



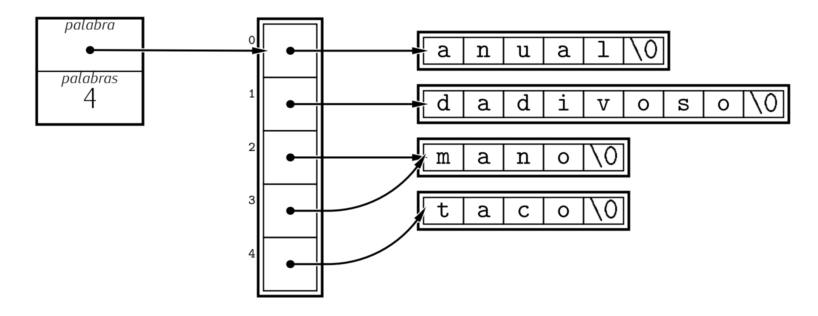
Una vez hemos redimensionado el vector de palabras, tenemos esto:



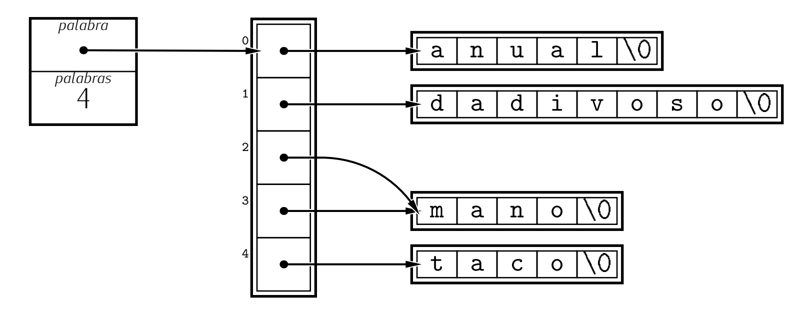
La nueva palabra debe insertarse en la posición de índice 2. El bucle ejecuta la asignación `d->palabra[j] = d->palabra[j-1]` para *j* tomando los valores 4 y 3. Cuando se ejecuta la iteración con *j* igual a 4, tenemos:



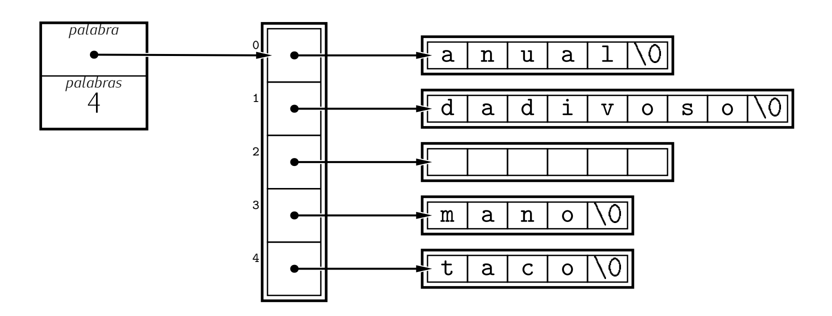
La ejecución de la asignación ha hecho que *d*->*palabra*[4] apunte al mismo lugar que *d*->*palabra*[3]. No hay problema alguno en que dos punteros apunten a un mismo bloque de memoria. En la siguiente iteración pasamos a esta otra situación:



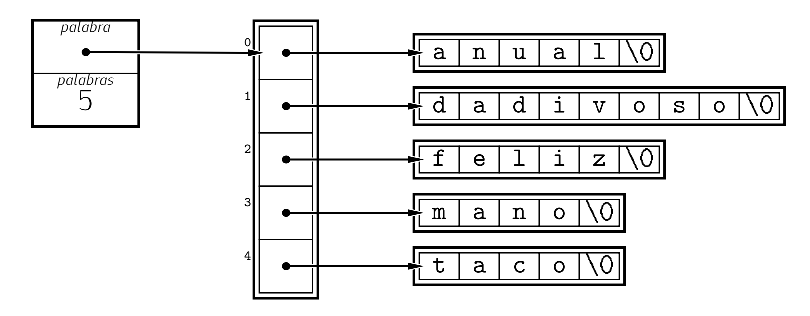
Podemos reordenar gráficamente los elementos, para ver que, efectivamente, estamos haciendo hueco para la nueva palabra:



El bucle ha acabado. Ahora se pide memoria para el puntero *d*->*palabra*[*i*] (siendo *i* igual a 2). Se piden 6 bytes («feliz» tiene 5 caracteres más el terminador nulo):



Y, finalmente, se copia en *d*->*palabra*[*i*] el contenido de *pal* con la función *strcpy* y se incrementa el valor de *d*->*palabras*:



Podemos ahora implementar una función de búsqueda de palabras más eficiente. Una primera idea consiste en buscar desde el principio y parar cuando se encuentre la palabra buscada o cuando se encuentre una palabra mayor (alfabéticamente) que la buscada. En este último caso sabremos que la palabra no existe. Pero aún hay una forma más eficiente de saber si una palabra está o no en una lista ordenada: mediante una *búsqueda dicotómica*.

```
int buscar_en_diccionario(struct Diccionario d, char pal[])
{
   int izquierda, centro, derecha;

   izquierda = 0;
   derecha = d.palabras;
   while (izquierda < derecha) {
      centro = (izquierda+derecha) / 2;
      if (strcmp(pal, d.palabra[centro]) == 0)
         return 1;
      else if (strcmp(pal, d.palabra[centro]) < 0)
         derecha = centro;
      else
         izquierda = centro+1;
   }
   return 0;
}
```

Podemos hacer una pequeña mejora para evitar el sobrecoste de llamar dos veces a la función *strcmp*:

```
int buscar_en_diccionario(struct Diccionario d, char pal[])
{
   int izquierda, centro, derecha, comparacion;

   izquierda = 0;
   derecha = d.palabras;
   while (izquierda < derecha) {
      centro = (izquierda+derecha) / 2;
      comparacion = strcmp(pal, d.palabra[centro]);
      if (comparacion == 0)
         return 1;
      else if (comparacion < 0)
         derecha = centro;
      else
         izquierda = centro+1;
   }
   return 0;
}
```

Juntemos todas las piezas y añadamos una función *main* que nos pida primero las palabras del diccionario y, a continuación, nos pida palabras que buscar en él:

diccionario.c — diccionario.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLON 80

struct Diccionario {
   char ** palabra;
```

```
  int palabras;
};

struct Diccionario crea_diccionario(void)
{
  struct Diccionario d;
  d.palabra = NULL;
  d.palabras = 0;
  return d;
}

void libera_diccionario(struct Diccionario * d)
{
  int i;

  if (d->palabra != NULL) {
    for (i=0; i<d->palabras; i++)
      free(d->palabra[i]);
    free(d->palabra);
    d->palabra = NULL;
    d->palabras = 0;
  }
}

void inserta_palabra_en_diccionario(struct Diccionario *d, char pal[])
{
  int i, j;

  for (i=0; i<d->palabras; i++) {
    if (strcmp(d->palabra[i], pal)==0) // Si ya está, no hay nada que hacer.
      return;
    if (strcmp(d->palabra[i], pal)>0)  // Aquí hemos encontrado su posición (orden alfabético)
      break;
  }

  /* Si llegamos aquí, la palabra no está y hay que insertarla en la posición i, desplazando
       antes el resto de palabras. */

  /* Reservamos espacio en la lista de palabras para una más. */
  d->palabra = realloc(d->palabra, (d->palabras+1) * sizeof(char *));
  /* Desplazamos el resto de palabras para que haya un hueco en el vector. */
  for (j=d->palabras; j>i; j--) {
    d->palabra[j] = malloc((strlen(d->palabra[j-1])+1)*sizeof(char));
    strcpy(d->palabra[j], d->palabra[j-1]);
    free(d->palabra[j-1]);
  }
  /* Y copiamos en su celda la nueva palabra */
  d->palabra[i] = malloc( (strlen(pal)+1) * sizeof(char) );
  strcpy(d->palabra[i], pal);
  d->palabras++;
}

int buscar_en_diccionario(struct Diccionario d, char pal[])
{
  int izquierda, centro, derecha, comparacion;

  izquierda = 0;
  derecha = d.palabras;
  while (izquierda < derecha) {
```

```
    centro = (izquierda+derecha) / 2;
    comparacion = strcmp(pal, d.palabra[centro]);
    if (comparacion == 0)
      return 1;
    else if (comparacion < 0)
      derecha = centro;
    else
      izquierda = centro+1;
  }
  return 0;
}

int main(void)
{
  struct Diccionario mi_diccionario;
  int num_palabras;
  char linea[MAXLON+1];

  mi_diccionario = crea_diccionario();

  printf("¿Cuántas palabras tendrá el diccionario?: ");
  gets(linea); sscanf(linea, "%d", &num_palabras);
  while (mi_diccionario.palabras != num_palabras) {
    printf("Palabra %d: ", mi_diccionario.palabras+1);
    gets(linea);
    inserta_palabra_en_diccionario(&mi_diccionario, linea);
  }

  do {
    printf("¿Qué palabra busco? (pulsa retorno para acabar): ");
    gets(linea);
    if (strlen(linea) > 0) {
      if (buscar_en_diccionario(mi_diccionario, linea))
        printf("Sí que está.\n");
      else
        printf("No está.\n");
    }
  } while(strlen(linea) > 0);

  libera_diccionario(&mi_diccionario);

  return 0;
}
```

EJERCICIOS

▶ 243 ¿Cuántas comparaciones se hacen en el peor de los casos en la búsqueda dicotómica de una palabra cualquiera en un diccionario con 8 palabras? ¿Y en un diccionario con 16 palabras? ¿Y en uno con 32? ¿Y en uno con 1024? ¿Y en uno con 1048576? (Nota: el valor 1048576 es igual a 2^{20}.)

▶ 244 Al insertar una nueva palabra en un diccionario hemos de comprobar si existía previamente y, si es una palabra nueva, averiguar en qué posición hay que insertarla. En la última versión presentada, esa búsqueda se efectúa recorriendo el diccionario palabra a palabra. Modifícala para que esa búsqueda sea dicotómica.

▶ 245 Diseña una función que funda dos diccionarios ordenados en uno sólo (también ordenado) que se devolverá como resultado. La fusión se hará de modo que las palabras que están repetidas en ambos diccionarios aparezcan una sóla vez en el diccionario final.

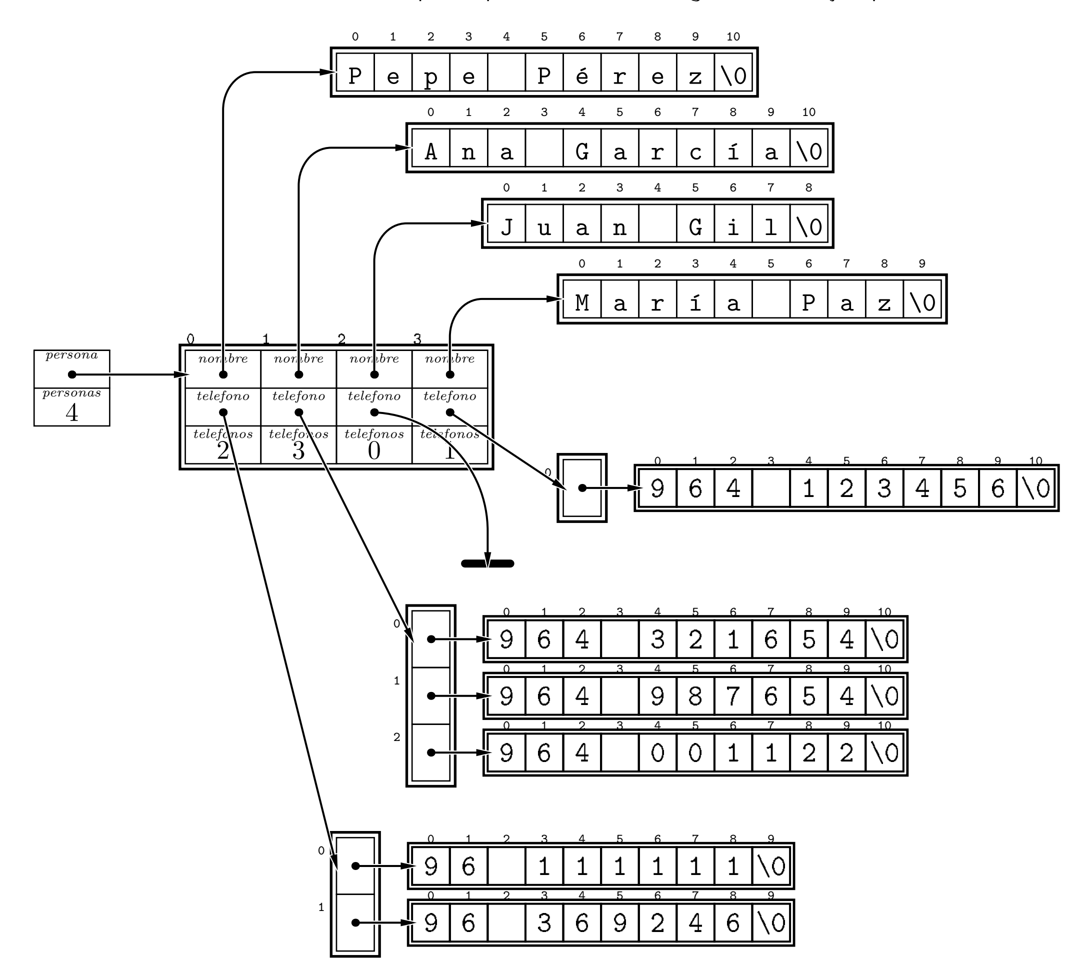
4.4.1. Una aplicación: una agenda telefónica

Vamos a desarrollar un ejemplo completo: una agenda telefónica. Utilizaremos vectores dinámicos en diferentes estructuras de datos del programa. Por ejemplo, el nombre de las personas registradas en la agenda y sus números de teléfonos consumirán exactamente la memoria que necesitan, sin desperdiciar un sólo byte. También el número de entradas en la agenda se adaptará a las necesidades reales de cada ejecución. El nombre de una persona no cambia durante la ejecución del programa, así que no necesitará redimensionamiento, pero la cantidad de números de teléfono de una misma persona sí (se pueden añadir números de teléfono a la entrada de una persona que ya ha sido dada de alta). Gestionaremos esa memoria con redimensionamiento, del mismo modo que usaremos redimensionamiento para gestionar el vector de entradas de la agenda: gastaremos exactamente la memoria que necesitemos para almacenar la información.

Aquí tienes un texto con el tipo de información que deseamos almacenar:

```
Pepe Pérez    96 111111    96 369246
Ana García   964 321654   964 987654   964 001122
Juan Gil
María Paz    964 123456
```

Para que te hagas una idea del montaje, te mostramos la representación gráfica de las estructuras de datos con las que representamos la agenda del ejemplo:



Empezaremos proporcionando «soporte» para el tipo de datos «entrada»: un nombre y un listado de teléfonos (un vector dinámico).

```
#include <stdio.h>
#include <stdlib.h>
```

```
/*************************************************************************
 * Entradas
 *************************************************************************/

struct Entrada {
  char * nombre;     // Nombre de la persona.
  char ** telefono; // Vector dinámico de números de teléfono.
  int telefonos;     // Número de elementos en el anterior vector.
};

void crea_entrada(struct Entrada * e, char * nombre)
  /* Inicializa una entrada con un nombre. La lista de teléfonos se pone a NULL. */
{
  /* Reservamos memoria para el nombre y efectuamos la asignación. */
  e->nombre = malloc((strlen(nombre)+1)*sizeof(char));
  strcpy(e->nombre, nombre);

  e->telefono = NULL;
  e->telefonos = 0;
}

void anyadir_telefono_a_entrada(struct Entrada * e, char * telefono)
{
  e->telefono = realloc(e->telefono, (e->telefonos+1)*sizeof(char *));
  e->telefono[e->telefonos] = malloc((strlen(telefono)+1)*sizeof(char));
  strcpy(e->telefono[e->telefonos], telefono);
  e->telefonos++;
}

void muestra_entrada(struct Entrada * e)
  // Podríamos haber pasado e por valor, pero resulta más eficiente (y no mucho más
  // incómodo) hacerlo por referencia: pasamos así sólo 4 bytes en lugar de 12.
{
  int i;

  printf("Nombre: %s\n", e->nombre);
  for(i=0; i<e->telefonos; i++)
    printf("    Teléfono %d: %s\n", i+1, e->telefono[i]);
}

void libera_entrada(struct Entrada * e)
{
  int i;

  free(e->nombre);
  for (i=0; i<e->telefonos; i++)
    free(e->telefono[i]);
  free(e->telefono);

  e->nombre = NULL;
  e->telefono = NULL;
  e->telefonos = 0;
}
```

. . . . EJERCICIOS . . . .

▶ **246** Modifica *anyadir_telefono_a_entrada* para que compruebe si el teléfono ya había sido dado de alta. En tal caso, la función dejará intacta la lista de teléfonos de esa entrada.

. . . .

Ya tenemos resuelta la gestión de entradas. Ocupémonos ahora del tipo agenda y de su gestión.

```
/************************************************************************
 * Agenda
 ************************************************************************/

struct Agenda {
  struct Entrada * persona;  /* Vector de entradas */
  int personas;              /* Número de entradas en el vector */
};

struct Agenda crea_agenda(void)
{
  struct Agenda a;

  a.persona = NULL;
  a.personas = 0;
  return a;
}

void anyadir_persona(struct Agenda * a, char * nombre)
{
  int i;

  /* Averiguar si ya tenemos una persona con ese nombre */
  for (i=0; i<a->personas; i++)
    if (strcmp(a->persona[i].nombre, nombre) == 0)
      return;

  /* Si llegamos aquí, es porque no teníamos registrada a esa persona. */
  a->persona = realloc(a->persona, (a->personas+1) * sizeof(struct Entrada));
  crea_entrada(&a->persona[a->personas], nombre);
  a->personas++;
}

void muestra_agenda(struct Agenda * a)
  // Pasamos a así por eficiencia.
{
  int i;

  for (i=0; i<a->personas; i++)
    muestra_entrada(&a->persona[i]);
}

struct Entrada * buscar_entrada_por_nombre(struct Agenda * a, char * nombre)
{
  int i;

  for (i=0; i<a->personas; i++)
    if (strcmp(a->persona[i].nombre, nombre)==0)
      return & a->persona[i];

  /* Si llegamos aquí, no lo hemos encontrado. Devolvemos NULL para indicar que no
      «conocemos» a esa persona. */
  return NULL;
}
```

```
void libera_agenda(struct Agenda * a)
{
  int i;

  for (i=0; i<a->personas; i++)
    libera_entrada(&a->persona[i]);
  free(a->persona);
  a->persona = NULL;
  a->personas = 0;
}
```

Fíjate en el prototipo de *buscar_entrada_por_nombre*: devuelve un puntero a un dato de tipo **struct** *Entrada*. Es un truco bastante utilizado. Si no existe una persona con el nombre indicado, se devuelve un puntero a NULL, y si existe, un puntero a esa persona. Ello nos permite, por ejemplo, mostrarla a continuación llamando a la función que muestra «entradas», pues espera un puntero a un **struct** *Entrada*. Ahora verás cómo lo hacemos en el programa principal.

Un lenguaje para cada tarea

Acabas de ver que el tratamiento de cadenas en C es bastante primitivo y nos obliga a estar pendientes de numerosos detalles. La memoria que ocupa cada cadena ha de ser explícitamente controlada por el programador y las operaciones que podemos hacer con ellas son, en principio, primitivas. Python, por contra, libera al programador de innumerables preocupaciones cuando trabaja con objetos como las cadenas o los vectores dinámicos (sus listas). Además, ofrece «de fábrica» numerosas utilidades para manipular cadenas (cortes, funciones del módulo *string*, etc.) y listas (método *append*, sentencia *del*, cortes, índices negativos, etc.). ¿Por qué no usamos siempre Python? Por eficiencia. C permite diseñar, por regla general, programas mucho más eficientes que sus equivalentes en Python. La mayor flexibilidad de Python tiene un precio.

Antes de programar una aplicación, hemos de preguntarnos si la eficiencia es un factor crítico. Un programa con formularios (con un interfaz gráfico) y/o accesos a una base de datos funcionará probablemente igual de rápido en C que en Python, ya que el cuello de botella de la ejecución lo introduce el propio usuario con su (lenta) velocidad de introducción de datos y/o el sistema de base de datos al acceder a la información. En tal caso, parece sensato escoger el lenguaje más flexible, el que permita desarrollar la aplicación con mayor facilidad. Un programa de cálculo matricial, un sistema de adquisición de imágenes para una cámara de vídeo digital, etc. han de ejecutarse a una velocidad que (probablemente) excluya a Python como lenguaje para la implementación.

Hay lenguajes de programación que combinan la eficiencia de C con parte de la flexibilidad de Python y pueden suponer una buena solución de compromiso en muchos casos. Entre ellos hemos de destacar el lenguaje C++, que estudiarás el próximo curso.

Y hay una opción adicional: implementar en el lenguaje eficiente las rutinas de cálculo pesadas y usarlas desde un lenguaje de programación flexible. Python ofrece un interfaz que permite el uso de módulos escritos en C o C++. Su uso no es trivial, pero hay herramientas como «SWIG» o «Boost.Python» que simplifican enormemente estas tareas.

Ya podemos escribir el programa principal.

agenda.c

```
#include <stdio.h>
#include <stdlib.h>

#define MAXLON_NOMBRE 200
#define MAXLON_TELEFONO 40
#define MAXLON_LINEA 80

```

```
enum { Ver=1, AltaPersona, AnyadirTelefono, Buscar, Salir };

```

⋮

```

/************************************************************************
 * Programa principal
 ************************************************************************/

int main(void)
{
  struct Agenda miagenda;
  struct Entrada * encontrada;
  int opcion;
  char nombre[MAXLON_NOMBRE+1];
  char telefono[MAXLON_TELEFONO+1];
  char linea[MAXLON_LINEA+1];

  miagenda = crea_agenda();

  do {
    printf("Menú:\n");
    printf("1) Ver contenido completo de la agenda.\n");
    printf("2) Dar de alta una persona.\n");
    printf("3) Añadir un teléfono.\n");
    printf("4) Buscar teléfonos de una persona.\n");
    printf("5) Salir.\n");
    printf("Opción: ");
    gets(linea); sscanf(linea, "%d", &opcion);

    switch(opcion) {

      case Ver:
        muestra_agenda(&miagenda);
        break;

      case AltaPersona:
        printf("Nombre: ");
        gets(nombre);
        anyadir_persona(&miagenda, nombre);
        break;

      case AnyadirTelefono:
        printf("Nombre: ");
        gets(nombre);
        encontrada = buscar_entrada_por_nombre(&miagenda, nombre);
        if (encontrada == NULL) {
          printf("No hay nadie llamado %s en la agenda.\n", nombre);
          printf("Por favor, dé antes de alta a %s.\n", nombre);
        }
        else {
          printf("Telefono: ");
          gets(telefono);
          anyadir_telefono_a_entrada(encontrada, telefono);
        }
        break;

      case Buscar:
        printf("Nombre: ");
```

```
          gets(nombre);
          encontrada = buscar_entrada_por_nombre(&miagenda, nombre);
          if (encontrada == NULL)
            printf("No␣hay␣nadie␣llamado␣%s␣en␣la␣agenda.\n", nombre);
          else
            muestra_entrada(encontrada);
          break;
      }
    } while (opcion != Salir);

    libera_agenda(&miagenda);

    return 0;
}
```

. . . . EJERCICIOS . . . .

▶ **247** Diseña una función que permita eliminar una entrada de la agenda a partir del nombre de una persona.

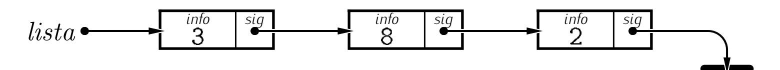
▶ **248** La agenda, tal y como la hemos implementado, está desordenada. Modifica el programa para que esté siempre ordenada.

. . . .

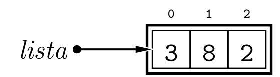
4.5. Introducción a la gestión de registros enlazados

Hemos aprendido a crear vectores dinámicos. Podemos crear secuencias de elementos de cualquier tamaño, aunque hemos visto que usar *realloc* para adaptar el número de elementos reservados a las necesidades de la aplicación es una posible fuente de ineficiencia, pues puede provocar la copia de grandes cantidades de memoria. En esta sección aprenderemos a crear *listas con registros enlazados*. Este tipo de listas ajustan su consumo de memoria al tamaño de la secuencia de datos que almacenan sin necesidad de llamar a *realloc*.

Una lista enlazada es una secuencia de registros unidos por punteros de manera que cada registro contiene un valor y nos indica cuál es el siguiente registro. Así pues, cada registro consta de uno o más campos con datos y un puntero: el que apunta al siguiente registro. El último registro de la secuencia apunta a… nada. Aparte, un «puntero maestro» apunta al primero de los registros de la secuencia. Fíjate en este gráfico para ir captando la idea:



Conceptualmente, es lo mismo que se ilustra en este gráfico:



Pero sólo conceptualmente. En la implementación, el nivel de complejidad al que nos enfrentamos es mucho mayor. Eso sí, a cambio ganaremos en flexibilidad y podremos ofrecer versiones eficientes de ciertas operaciones sobre listas de valores que implementadas con vectores dinámicos serían muy costosas. Por otra parte, aprender estas técnicas supone una inversión a largo plazo: muchas estructuras dinámicas que estudiarás en cursos de Estructuras de Datos se basan en el uso de registros enlazados y, aunque mucho más complejas que las simples secuencias de valores, usan las mismas técnicas básicas de gestión de memoria que aprenderás ahora.

Redimensionamiento con holgura

Utilizamos *realloc* para aumentar celda a celda la reserva de memoria de los vectores que necesitan crecer. Estos redimensionamientos, tan ajustados a las necesidades exactas de cada momento, pasan factura: cada *realloc* es potencialmente lento, pues sabes que puede disparar la copia de un bloque de memoria. Una modo de paliar este problema consiste en crecer varias celdas cuando nos quedamos cortos de memoria en un vector. Un campo adicional en el registro, llamémosle *capacidad*, indica cuántas celdas tiene reservadas el vector y otro campo, digamos, *talla*, indica cuántas de ellas están realmente ocupadas. Por ejemplo, este vector, en el que sólo ocupamos de momento tres celdas (las marcadas en negro), tendría *talla* 3 y *capacidad* 5:



Cuando necesitamos escribir un nuevo dato en una celda adicional, comprobamos si *talla* es menor o igual que *capacidad*. En tal caso, no redimensionamos el vector: incrementamos el valor de *talla*. Pero en caso contrario, nos curamos en salud y redimensionamos pidiendo memoria para, pongamos, 10 celdas más (y, consecuentemente, incrementamos *capacidad* en 10 unidades). Así reducimos las llamadas a *realloc* a la décima parte. Incrementar un número fijo de celdas no es la única estrategia posible. Otra aproximación consiste en duplicar la capacidad cada vez que se precisa agrandar el vector. De este modo, el número de llamadas a *realloc* es proporcional al logaritmo en base 2 del número de celdas del vector.

```c
struct VDH { // vector Dinámico con Holgura (VDH)
  int * dato;
  int talla, capacidad;
};

struct VDH crea_VDH(void) {
  struct VDH vdh;
  vdh.dato = malloc(1*sizeof(int)); vdh.talla = 0; vdh.capacidad = 1;
  return vdh;
}

void anyade_dato(struct VDH * vdh, int undato)
{
  if (vdh->talla == vdh->capacidad) {
    vdh->capacidad *= 2;
    vdh->dato = realloc(vdh->dato, vdh->capacidad * sizeof(int));
  }
  vdh->dato[vdh->talla++] = undato;
}

void elimina_ultimo(struct VDH * vdh)
{
  if (vdh->talla < vdh->capacidad/2 && vdh->capacidad > 0) {
    vdh->capacidad /= 2;
    vdh->dato = realloc(vdh->dato, vdh->capacidad * sizeof(int));
  }
  vdh->talla--;
}
```

Ciertamente, esta aproximación «desperdicia» memoria, pero en una cantidad que puede ser tolerable para nuestra aplicación. Python usa internamente una variante de esta técnica al modificar la talla de una lista con el método *append*.

Para ir aprendiendo a usar estas técnicas, gestionaremos ahora una lista con registros enlazados. La manejaremos directamente, desde un programa principal, y nos centraremos en la realización de una serie de acciones que parten de un estado de la lista y la dejan en otro estado diferente. Ilustraremos cada una de las acciones mostrando con todo detalle qué ocurre paso a paso. En el siguiente apartado «encapsularemos» cada acción elemental (añadir elemento, borrar elemento, etc.) en una función independiente.

4.5.1. Definición y creación de la lista

Vamos a crear una *lista de enteros*. Empezamos por definir el tipo de los registros que componen la lista:

```
struct Nodo {
  int info;
  struct Nodo * sig;
};
```

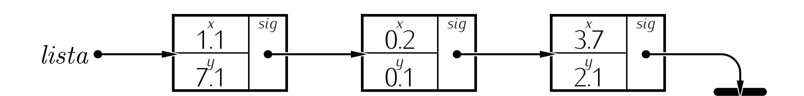
Un registro de tipo **struct** *Nodo* consta de dos elementos:

- un entero llamado *info*, que es la información que realmente nos importa,
- y un puntero a un elemento que es... ¡otro **struct** *Nodo*! (Observa que hay cierto nivel de recursión o autoreferencia en la definición: un **struct** *Nodo* contiene un puntero a un **struct** *Nodo*. Por esta razón, las estructuras que vamos a estudiar se denominan a veces *estructuras recursivas*.)

Si quisiéramos manejar una lista de puntos en el plano, tendríamos dos opciones:

1. definir un registro con varios campos para la información relevante:

```
struct Nodo {
  float x ;
  float y ;
  struct Nodo * sig;
};
```



2. definir un tipo adicional y utilizar un único campo de dicho tipo:

```
struct Punto {
  float x;
  float y;
};

struct Nodo {
  struct Punto info;
  struct Nodo * sig;
};
```

Cualquiera de las dos opciones es válida, si bien la segunda es más elegante.

Volvamos al estudio de nuestra lista de enteros. Creemos ahora el «puntero maestro», aquél en el que empieza la lista de enteros:

```
int main(void)
{
  struct Nodo * lista;

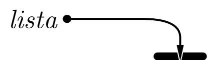
  ...
```

No es más que un puntero a un elemento de tipo **struct** *Nodo*. Inicialmente, la lista está vacía. Hemos de indicarlo explícitamente así:

```
int main(void)
{
  struct Nodo * lista = NULL;

  ...
```

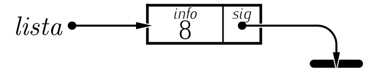
Tenemos ahora esta situación:



O sea, *lista* no contiene nada, está vacía.

4.5.2. Adición de nodos (por cabeza)

Empezaremos añadiendo un nodo a la lista. Nuestro objetivo es pasar de la lista anterior a esta otra:

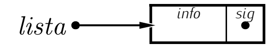


¿Cómo creamos el nuevo registro? Con *malloc*:

```
int main(void)
{
  struct Nodo * lista = NULL;

  lista = malloc( sizeof(struct Nodo) );
  ...
```

Éste es el resultado:



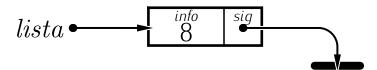
Ya tenemos el primer nodo de la lista, pero sus campos aún no tienen los valores que deben tener finalmente. Lo hemos representado gráficamente dejando el campo *info* en blanco y sin poner una flecha que salga del campo *sig*.

Por una parte, el campo *info* debería contener el valor 8, y por otra, el campo *sig* debería apuntar a NULL:

```
int main(void)
{
  struct Nodo * lista = NULL;

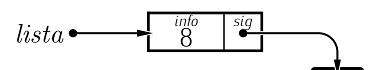
  lista = malloc( sizeof(struct Nodo) );
  lista->info = 8;
  lista->sig = NULL;
  ...
```

No debe sorprenderte el uso del operador -> en las asignaciones a campos del registro. La variable *lista* es de tipo **struct** *Nodo* *, es decir, es un puntero, y el operador -> permite acceder al campo de un registro apuntado por un puntero. He aquí el resultado:

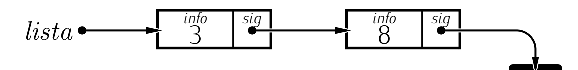


Ya tenemos una lista con un único elemento.

Vamos a añadir un nuevo nodo a la lista, uno que contenga el valor 3 y que ubicaremos justo al principio de la lista, delante del nodo que contiene el valor 8. O sea, partimos de esta situación:



y queremos llegar a esta otra:

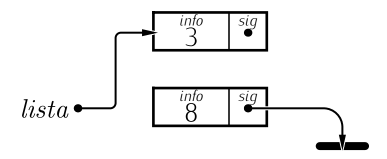


En primer lugar, hemos de crear un nuevo nodo al que deberá apuntar *lista*. El campo *sig* del nuevo nodo, por su parte, debería apuntar al nodo que contiene el valor 8. Empecemos por la petición de un nuevo nodo que, ya que debe ser apuntado por *lista*, podemos pedir y rellenar así:

```
int main(void)
{
  struct Nodo * lista = NULL;

  ...
  lista = malloc( sizeof(struct Nodo) ) ;
  lista->info = 3;
  lista->sig = ??? ; // No sabemos cómo expresar esta asignación.
  ...
```

¡Algo ha ido mal! ¿Cómo podemos asignar a *lista*->*sig* la dirección del siguiente nodo con valor 8? La situación en la que nos encontramos se puede representar así:

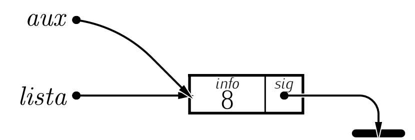


¡No somos capaces de acceder al nodo que contiene el valor 8! Es lo que denominamos una *pérdida de referencia*, un grave error en nuestro programa que nos imposibilita seguir construyendo la lista. Si no podemos acceder a un bloque de memoria que hemos pedido con *malloc*, tampoco podremos liberarlo luego con *free*. Cuando se produce una pérdida de referencia hay, pues, una *fuga de memoria*: pedimos memoria al ordenador y no somos capaces de liberarla cuando dejamos de necesitarla. Un programa con fugas de memoria corre el riesgo de consumir toda la memoria disponible en el ordenador. Hemos de estar siempre atentos para evitar pérdidas de referencia. Es uno de los mayores peligros del trabajo con memoria dinámica.

¿Cómo podemos evitar la pérdida de referencia? Muy fácil: con un puntero auxiliar.

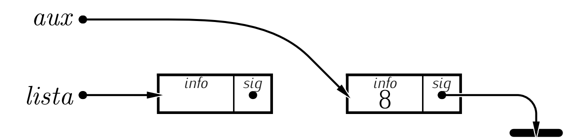
```
1  int main(void)
2  {
3    struct Nodo * lista = NULL, * aux ;
4
5    ...
6    aux = lista ;
7    lista = malloc( sizeof(struct Nodo) ) ;
8    lista->info = 3;
9    lista->sig = aux ;
10   ...
```

La declaración de la línea 3 es curiosa. Cuando declaras dos o más punteros en una sola línea, has de poner el asterisco delante del identificador de cada puntero. En una línea de declaración que empieza por la palabra **int** puedes declarar punteros a enteros y enteros, según precedas los respectivos identificadores con asterisco o no. Detengámonos un momento para considerar el estado de la memoria justo después de ejecutarse la línea 6, que reza «*aux* = *lista*»:

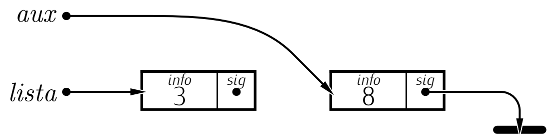


El efecto de la línea 6 es que tanto *aux* como *lista* apuntan al mismo registro. *La asignación de un puntero a otro hace que ambos apunten al mismo elemento*. Recuerda que un puntero no es más que una dirección de memoria y que copiar un puntero a otro hace que ambos contengan la misma dirección de memoria, es decir, que ambos apunten al mismo lugar.

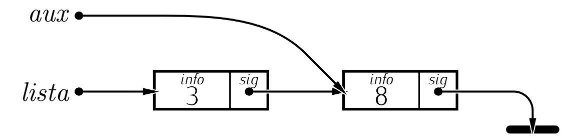
Sigamos con nuestra traza. Veamos cómo queda la memoria justo después de ejecutar la línea 7, que dice «*lista* = *malloc*(**sizeof**(**struct** *Nodo*))»:



La línea 8, que dice «*lista*->*info* = 3», asigna al campo *info* del nuevo nodo (apuntado por *lista*) el valor 3:



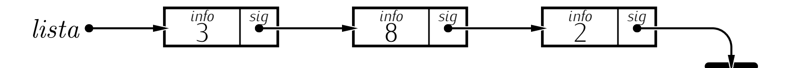
La lista aún no está completa, pero observa que no hemos perdido la referencia al último fragmento de la lista. El puntero *aux* la mantiene. Nos queda por ejecutar la línea 9, que efectúa la asignación «*lista*->*sig* = *aux*» y enlaza así el campo *sig* del primer nodo con el segundo nodo, el apuntado por *aux*. Tras ejecutarla tenemos:



¡Perfecto! ¿Seguro? ¿Y qué hace *aux* apuntando aún a la lista? La verdad, nos da igual. *Lo importante es que los nodos que hay enlazados desde lista formen la lista que queríamos construir*. No importa cómo quedan los punteros auxiliares: una vez han desempeñado su función en la construcción de la lista, son supérfluos. Si te quedas más tranquilo, puedes añadir una línea con *aux* = NULL al final del programa para que *aux* no quede apuntando a un nodo de la lista, pero, repetimos, es innecesario.

4.5.3. Adición de un nodo (por cola)

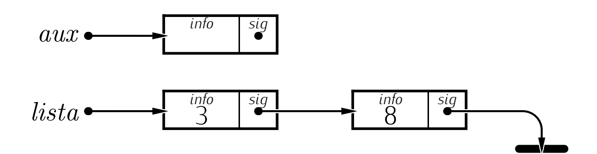
Marquémonos un nuevo objetivo. Intentemos añadir un nuevo nodo *al final* de la lista. Es decir, partiendo de la última lista, intentemos obtener esta otra:



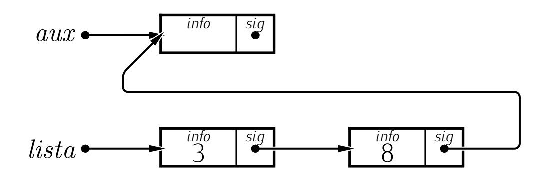
¿Qué hemos de hacer? Para empezar, pedir un nuevo nodo, sólo que esta vez no estará apuntado por *lista*, sino por el que hasta ahora era el último nodo de la lista. De momento, lo mantendremos apuntado por un puntero auxiliar. Después, accederemos de algún modo al campo *sig* del último nodo de la lista (el que tiene valor 8) y haremos que apunte al nuevo nodo. Finalmente, haremos que el nuevo nodo contenga el valor 2 y que tenga como siguiente nodo a `NULL`. Intentémoslo:

```
1  int main(void)
2  {
3    struct Nodo * lista = NULL, * aux;
4
5    ...
6    aux = malloc( sizeof(struct Nodo) ) ;
7    lista->sig->sig = aux ;
8    aux->info = 2 ;
9    aux->sig = NULL ;
10
11   return 0;
12 }
```

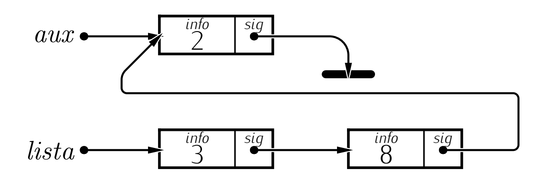
Veamos cómo queda la memoria paso a paso. Tras ejecutar la línea 6 tenemos:



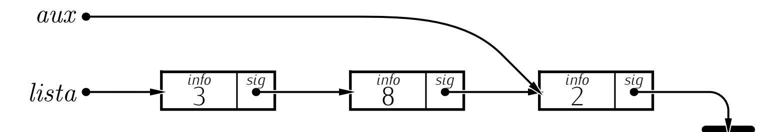
O sea, la lista que «cuelga» de *lista* sigue igual, pero ahora *aux* apunta a un nuevo nodo. Pasemos a estudiar la línea 7, que parece complicada porque contiene varias aplicaciones del operador `->`. Esa línea reza así: *lista*`->`*sig*`->`*sig* = *aux*. Vamos a ver qué significa leyéndola de izquierda a derecha. Si *lista* es un puntero, y *lista*`->`*sig* es el campo *sig del primer nodo*, que es otro puntero, entonces *lista*`->`*sig*`->`*sig* es el campo *sig del segundo nodo*, que es otro puntero. Si a ese puntero le asignamos *aux*, el campo *sig* del segundo nodo apunta a donde apuntará *aux*. Aquí tienes el resultado:



Aún no hemos acabado. Una vez hayamos ejecutado las líneas 8 y 9, el trabajo estará completo:



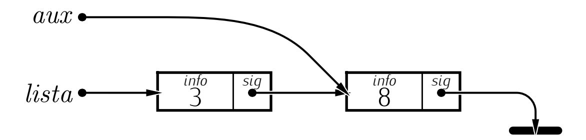
¿Y es éso lo que buscábamos? Sí. Reordenemos gráficamente los diferentes componentes para que su disposición en la imagen se asemeje más a lo que esperábamos:



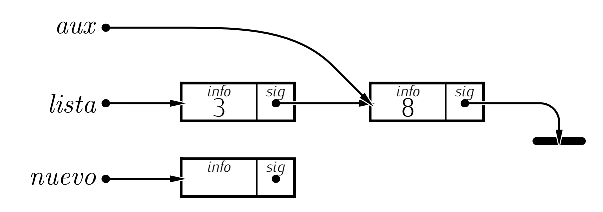
Ahora queda más claro que, efectivamente, hemos conseguido el objetivo. Esta figura y la anterior son absolutamente equivalentes.

Aún hay algo en nuestro programa poco elegante: la asignación «*lista*->*sig*->*sig* = *aux*» es complicada de entender y da pie a un método de adición por el final muy poco «extensible». ¿Qué queremos decir con esto último? Que si ahora queremos añadir a la lista de 3 nodos un cuarto nodo, tendremos que hacer «*lista*->*sig*->*sig*->*sig* = *aux*». Y si quisiéramos añadir un quinto, «*lista*->*sig*->*sig*->*sig*->*sig* = *aux*» Imagina que la lista tiene 100 o 200 elementos. ¡Menuda complicación proceder así para añadir por el final! ¿No podemos expresar la idea «añadir por el final» de un modo más elegante y general? Sí. Podemos hacer lo siguiente:

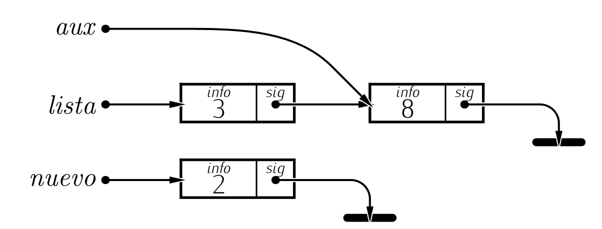
1. buscar el último elemento con un bucle y mantenerlo referenciado con un puntero auxiliar, digamos *aux*;



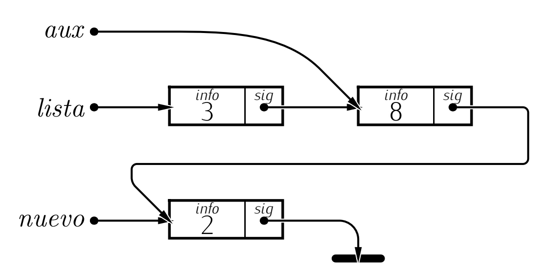
2. pedir un nodo nuevo y mantenerlo apuntado con otro puntero auxiliar, digamos *nuevo*;



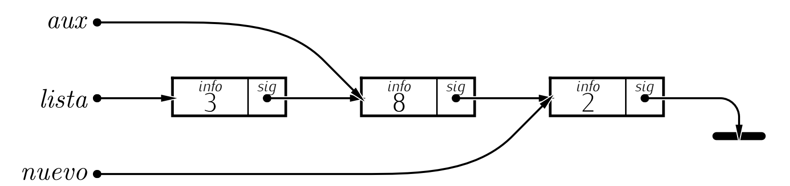
3. escribir en el nodo apuntado por *nuevo* el nuevo dato y hacer que su campo *sig* apunte a `NULL`;



4. hacer que el nodo apuntado por *aux* tenga como siguiente nodo al nodo apuntado por *nuevo*.



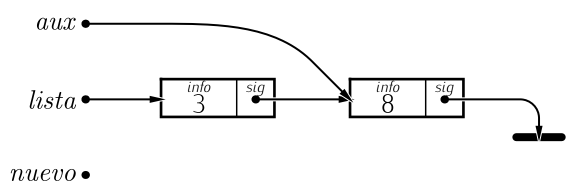
Lo que es equivalente a este otro gráfico en el que, sencillamente, hemos reorganizado la disposición de los diferentes elementos:



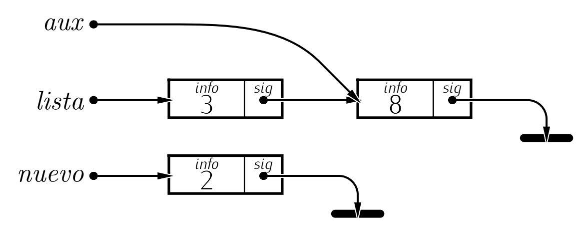
Modifiquemos el último programa para expresar esta idea:

```
1  int main(void)
2  {
3    struct Nodo * lista = NULL, * aux, * nuevo;
4
5    ...
6    aux = lista;
7    while (aux->sig != NULL)
8      aux = aux->sig;
9    nuevo = malloc( sizeof(struct Nodo) );
10   nuevo->info = 2;
11   nuevo->sig = NULL;
12   aux->sig = nuevo;
13
14   return 0;
15 }
```

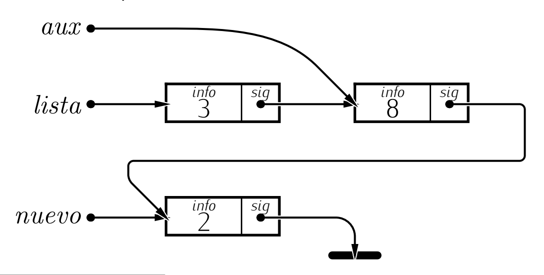
La inicialización y el bucle de las líneas 6–8 buscan al último nodo de la lista y lo mantienen apuntado con *aux*. El último nodo se distingue porque al llegar a él, *aux*->*sig* es NULL, de ahí la condición del bucle. No importa cuán larga sea la lista: tanto si tiene 1 elemento como si tiene 200, *aux* acaba apuntando al último de ellos.[5] Si partimos de una lista con dos elementos, éste es el resultado de ejecutar el bucle:



Las líneas 9–11 dejan el estado de la memoria así:



Finalmente, la línea 12 completa la adición del nodo:



[5]Aunque falla en un caso: si la lista está inicialmente vacía. Estudiaremos este problema y su solución más adelante.

Y ya está. Eso es lo que buscábamos.

La inicialización y el bucle de las líneas 6–8 se pueden expresar en C de una forma mucho más compacta usando una estructura **for**:

```
int main(void)
{
  struct Nodo * lista = NULL, * aux, * nuevo;

  ...
  for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
  nuevo = malloc( sizeof(struct Nodo) );
  nuevo->info = 2;
  nuevo->sig = NULL;
  aux->sig = nuevo;

  return 0;
}
```

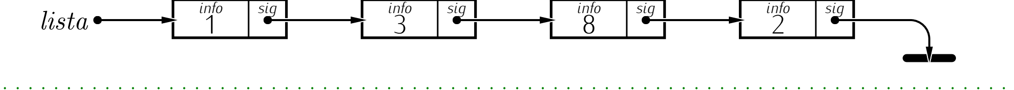
Observa que el punto y coma que aparece al final del bucle **for** hace que no tenga sentencia alguna en su bloque:

```
  for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
```

El bucle se limita a «desplazar» el puntero *aux* hasta que apunte al último elemento de la lista. Esta expresión del bucle que busca el elemento final es más propia de la programación C, más idiomática.

. . . . EJERCICIOS . . . .

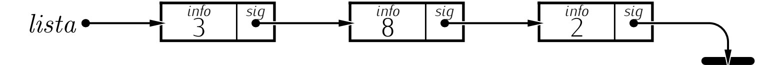
▶ 249 Hemos diseñado un método (que mejoraremos en el siguiente apartado) que permite insertar elementos por el final de una lista y hemos necesitado un bucle. ¿Hará falta un bucle para insertar un elemento por delante en una lista cualquiera? ¿Cómo harías para convertir la última lista en esta otra?:



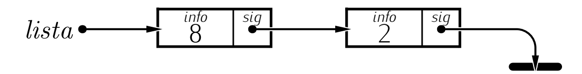
. . . .

4.5.4. Borrado de la cabeza

Vamos a aprender ahora a borrar elementos de una lista. Empezaremos por ver cómo eliminar el primer elemento de una lista. Nuestro objetivo es, partiendo de esta lista:



llegar a esta otra:



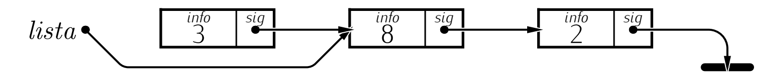
Como lo que deseamos es que *lista* pase a apuntar al segundo elemento de la lista, podríamos diseñar una aproximación directa modificando el valor de *lista*:

```
int main(void)
{
  struct Nodo * lista = NULL, * aux, * nuevo;

  ...
  lista = lista->sig; // ¡Mal! Se pierde la referencia a la cabeza original de la lista.

  return 0;
}
```

El efecto obtenido por esa acción es éste:



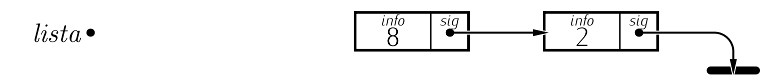
> ***Fugas de memoria, colapsos y recogida de basura***
>
> Muchos programas funcionan correctamente... durante un rato. Cuando llevan un tiempo ejecutándose, sin embargo, el ordenador empieza a ralentizarse sin explicación aparente y la memoria del ordenador se va agotando. Una de las razones para que esto ocurra son las fugas de memoria. Si el programa pide bloques de memoria con *malloc* y no los libera con *free*, irá consumiendo más y más memoria irremediablemente. Llegará un momento en que no quede apenas memoria libre y la que quede, estará muy fragmentada, así que las peticiones a *malloc* costarán más y más tiempo en ser satisfechas... ¡si es que pueden ser satisfechas! La saturación de la memoria provocada por la fuga acabará colapsando al ordenador y, en algunos sistemas operativos, obligando a reiniciar la máquina.
>
> El principal problema con las fugas de memoria es lo difíciles de detectar que resultan. Si pruebas el programa en un ordenador con mucha memoria, puede que no llegues a apreciar efecto negativo alguno al efectuar pruebas. Dar por bueno un programa erróneo es, naturalmente, peor que saber que el programa aún no es correcto.
>
> Los lenguajes de programación modernos suelen evitar las fugas de memoria proporcionando *recogida de basura* (del inglés *garbage collection*) automática. Los sistemas de recogida de basura detectan las pérdidas de referencia (origen de las fugas de memoria) y llaman automáticamente a *free* por nosotros. El programador sólo escribe llamadas a *malloc* (o la función/mecanismo equivalente en su lenguaje) y el sistema se encarga de marcar como disponibles los bloques de memoria no referenciados. Lenguajes como Python, Perl, Java, Ruby, Tcl y un largo etcétera tiene recogida de basura automática, aunque todos deben la idea a Lisp un lenguaje diseñado en los años 50 («¡!!!) que ya incorporaba esta «avanzada» característica.

Efectivamente, hemos conseguido que la lista apuntada por *lista* sea lo que pretendíamos, pero hemos perdido la referencia a un nodo (el que hasta ahora era el primero) y ya no podemos liberarlo. Hemos provocado una fuga de memoria.

Para liberar un bloque de memoria hemos de llamar a *free* con el puntero que apunta a la dirección en la que empieza el bloque. Nuestro bloque está apuntado por *lista*, así que podríamos pensar que la solución es trivial y que bastaría con llamar a *free* antes de modificar *lista*:

```
1  int main(void)
2  {
3    struct Nodo * lista = NULL, * aux, * nuevo;
4
5    ...
6    free(lista);
7    lista = lista->sig; // ¡Mal! lista no apunta a una zona de memoria válida.
8
9    return 0;
10 }
```

Pero, claro, no iba a resultar tan sencillo. ¡La línea 7, que dice «*lista* = *lista*->*sig*», no puede ejecutarse! Tan pronto hemos ejecutado la línea 6, tenemos otra fuga de memoria:



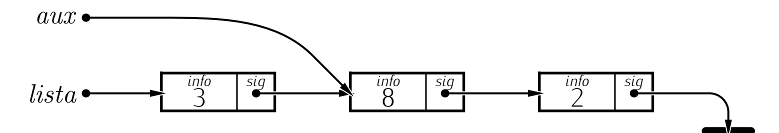
O sea, hemos liberado correctamente el primer nodo, pero ahora hemos perdido la referencia al resto de nodos y el valor de *lista*->*sig* está indefinido. ¿Cómo podemos arreglar esto? Si no liberamos memoria, hay una fuga, y si la liberamos perdemos la referencia al resto de la lista. La solución es sencilla: guardamos una referencia al resto de la lista con un puntero auxiliar cuando aún estamos a tiempo.

```
int main(void)
{
  struct Nodo * lista = NULL, * aux, * nuevo;

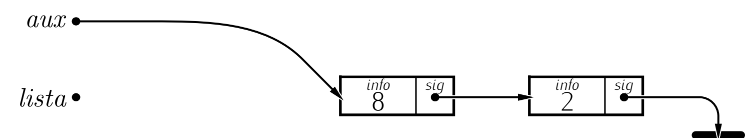
  ...
  aux = lista->sig;
  free(lista);
  lista = aux;

  return 0;
}
```

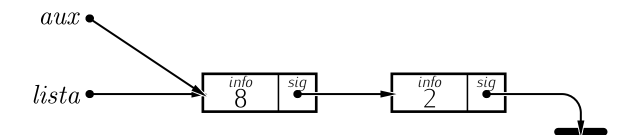
Ahora sí. Veamos paso a paso qué hacen las últimas tres líneas del programa. La asignación *aux* = *lista*->*sig* introduce una referencia al segundo nodo:



Al ejecutar *free*(*lista*), pasamos a esta otra situación:



No hay problema. Seguimos sabiendo dónde está el resto de la lista: «cuelga» de *aux*. Así pues, podemos llegar al resultado deseado con la asignación *lista* = *aux*:



¿Vas viendo ya el tipo de problemas al que nos enfrentamos con la gestión de listas? Los siguientes apartados te presentan funciones capaces de inicializar listas, de insertar, borrar y encontrar elementos, de mantener listas ordenadas, etc. Cada apartado te presentará una variante de las listas enlazadas con diferentes prestaciones que permiten elegir soluciones de compromiso entre velocidad de ciertas operaciones, consumo de memoria y complicación de la implementación.

4.6. Listas con enlace simple

Vamos a desarrollar un módulo que permita manejar listas de enteros. En el fichero de cabecera declararemos los tipos de datos básicos:

```
lista.h

struct Nodo {
  int info;
  struct Nodo * sig;
};
```

Como ya dijimos, este tipo de nodo sólo alberga un número entero. Si necesitásemos una lista de **float** deberíamos cambiar el tipo del valor del campo *info*. Y si quisiésemos una lista de «personas», podríamos añadir varios campos a **struct** *Nodo* (uno para el nombre, otro para la edad, etc.) o declarar *info* como de un tipo **struct** *Persona* definido previamente por nosotros.

Una lista es un puntero a un **struct** *Nodo*, pero cuesta poco definir un nuevo tipo para referirnos con mayor brevedad al tipo «lista»:

lista.h

```
...
typedef struct Nodo * TipoLista;
```

Ahora, podemos declarar una lista como **struct** *Nodo* * o como **TipoLista**, indistintamente. Por claridad, nos referiremos al tipo de una lista con **TipoLista** y al de un puntero a un nodo cualquiera con **struct** *Nodo* *, pero no olvides que ambos tipos son equivalentes.

Definición de struct con typedef

Hay quienes, para evitar la escritura repetida de la palabra **struct**, recurren a la inmediata creación de un nuevo tipo tan pronto se define el **struct**. Este código, por ejemplo, hace eso:

```
typedef struct Nodo {
  int info;
  struct Nodo * sig;
} TipoNodo;
```

Como **struct** *Nodo* y **TipoNodo** son sinónimos, pronto se intenta definir la estructura así:

```
typedef struct Nodo {
  int info;
  TipoNodo * sig; // ¡Mal!
} TipoNodo;
```

Pero el compilador emite un aviso de error. La razón es simple: la primera aparición de la palabra **TipoNodo** tiene lugar antes de su propia definición.

4.6.1. Creación de lista vacía

Nuestra primera función creará una lista vacía. El prototipo de la función, que declaramos en la cabecera `lista.h`, es éste:

lista.h

```
...
extern TipoLista lista_vacia(void);
```

y la implementación, que proporcionamos en una unidad de compilación `lista.c`, resulta trivial:

lista.c

```
#include <stdlib.h>
#include "lista.h"

TipoLista lista_vacia(void)
{
  return NULL;
}
```

La forma de uso es muy sencilla:

```
#include <stdlib.h>
#include "lista.h"

int main(void)
{
  TipoLista lista;

  lista = lista_vacia();

  return 0;
}
```

Ciertamente podríamos haber hecho *lista* = `NULL`, sin más, pero queda más elegante proporcionar funciones para cada una de las operaciones básicas que ofrece una lista, y crear una lista vacía es una operación básica.

4.6.2. ¿Lista vacía?

Nos vendrá bien disponer de una función que devuelva cierto o falso en función de si la lista está vacía o no. El prototipo de la función es:

`lista.h`

```
...
extern int es_lista_vacia(TipoLista lista);
```

y su implementación, muy sencilla:

`lista.c`

```
int es_lista_vacia(TipoLista lista)
{
  return lista == NULL;
}
```

4.6.3. Inserción por cabeza

Ahora vamos a crear una función que inserta un elemento en una lista por la cabeza, es decir, haciendo que el nuevo nodo sea el primero de la lista. Antes, veamos cuál es el prototipo de la función:

`lista.h`

```
...
extern TipoLista inserta_por_cabeza(TipoLista lista, int valor);
```

La forma de uso de la función será ésta:

`miprograma.c`

```
#include "lista.h"

int main(void)
{
  TipoLista lista;

  lista = lista_vacia();
  lista = inserta_por_cabeza(lista, 2);
  lista = inserta_por_cabeza(lista, 8);
  lista = inserta_por_cabeza(lista, 3);
  ...
  return 0;
}
```

o, equivalentemente, esta otra:

miprograma.c

```
#include "lista.h"

int main(void)
{
  TipoLista lista;

  lista = inserta_por_cabeza(inserta_por_cabeza(
                inserta_por_cabeza(lista_vacia(),2),8),3);
  ...
  return 0;
}
```

Vamos con la implementación de la función. La función debe empezar pidiendo un nuevo nodo para el número que queremos insertar.

lista.c

```
TipoLista inserta_por_cabeza(TipoLista lista, int valor)
{
  struct Nodo * nuevo = malloc(sizeof(struct Nodo));

  nuevo->info = valor;
  ...
}
```

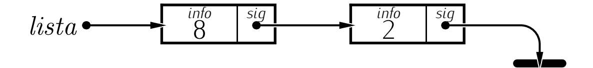
Ahora hemos de pensar un poco. Si *lista* va a tener como primer elemento a *nuevo*, ¿podemos enlazar directamente *lista* con *nuevo*?

lista.c

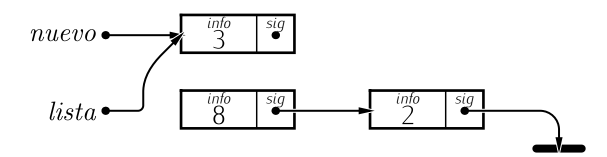
```
TipoLista inserta_por_cabeza(TipoLista lista, int valor)@mal
{
  struct Nodo * nuevo = malloc(sizeof(struct Nodo));

  nuevo->info = valor;
  lista = nuevo;
  ...
}
```

La respuesta es no. Aún no podemos. Si lo hacemos, no hay forma de enlazar *nuevo->sig* con lo que era la lista anteriormente. Hemos perdido la referencia a la lista original. Veámoslo con un ejemplo. Imagina una lista como ésta:



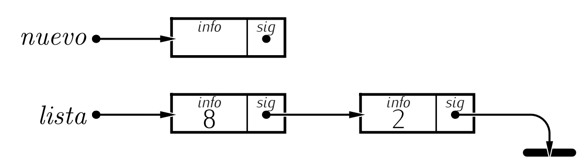
La ejecución de la función (incompleta) con *valor* igual a 3 nos lleva a esta otra situación:



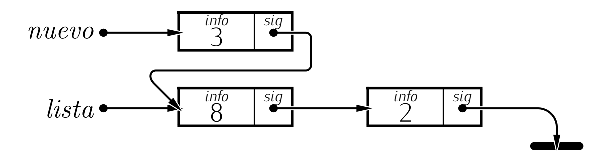
Hemos perdido la referencia a la «vieja» lista. Una solución sencilla consiste en, antes de modificar *lista*, asignar a *nuevo->sig* el valor de *lista*:

```
1 TipoLista inserta_por_cabeza(TipoLista lista, int valor)
2 {
3   struct Nodo * nuevo = malloc(sizeof(struct Nodo));
4
5   nuevo->info = valor;
6   nuevo->sig = lista;
7   lista = nuevo;
8   return lista;
9 }
```

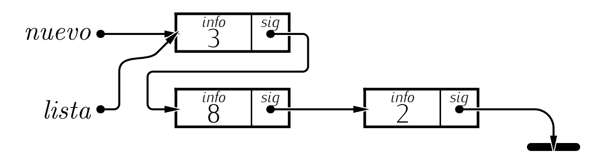
Tras ejecutarse la línea 3, tenemos:



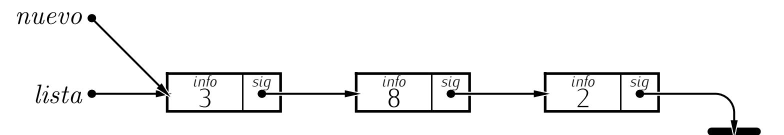
Las líneas 5 y 6 modifican los campos del nodo apuntado por *nuevo*:



Finalmente, la línea 7 hace que *lista* apunte a donde *nuevo* apunta. El resultado final es éste:

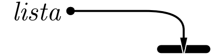


Sólo resta redisponer gráficamente la lista para que no quepa duda de la corrección de la solución:

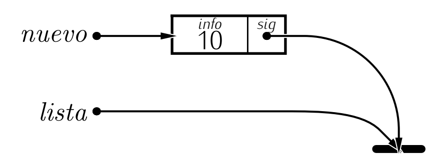


Hemos visto, pues, que el método es correcto cuando la lista no está vacía. ¿Lo será también si suministramos una lista vacía? *La lista vacía es un caso especial para el que siempre deberemos considerar la validez de nuestros métodos.*

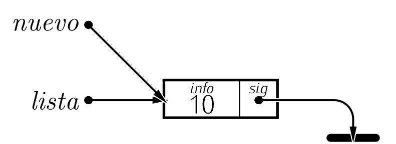
Hagamos una comprobación gráfica. Si partimos de esta lista:



y ejecutamos la función (con *valor* igual a 10, por ejemplo), pasaremos momentáneamente por esta situación:



y llegaremos, al final, a esta otra:



Ha funcionado correctamente. No tendremos tanta suerte con todas las funciones que vamos a diseñar.

4.6.4. Longitud de una lista

Nos interesa conocer ahora la longitud de una lista. La función que diseñaremos recibe una lista y devuelve un entero:

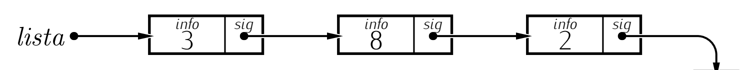
lista.h
```
...
extern int longitud_lista(TipoLista lista);
```

La implementación se basa en recorrer toda la lista con un bucle que desplace un puntero hasta llegar a `NULL`. Con cada salto de nodo a nodo, incrementaremos un contador cuyo valor final será devuelto por la función:

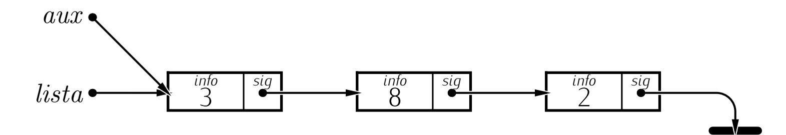
lista.c
```
int longitud_lista(TipoLista lista)
{
  struct Nodo * aux;
  int contador = 0;

  for (aux = lista; aux != NULL; aux = aux->sig)
    contador++;
  return contador;
}
```

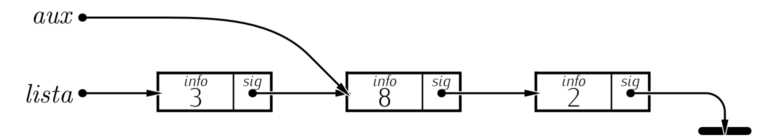
Hagamos una pequeña traza. Si recibimos esta lista:



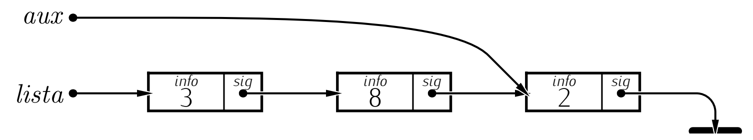
la variable contador empieza valiendo 0 y el bucle inicializa *aux* haciendo que apunte al primer elemento:



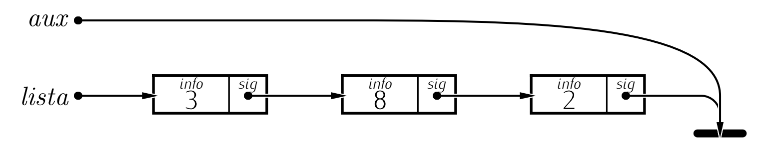
En la primera iteración, *contador* se incrementa en una unidad y *aux* pasa a apuntar al segundo nodo:



Acto seguido, en la segunda iteración, *contador* pasa a valer 2 y *aux* pasa a apuntar al tercer nodo:



Finalmente, *contador* vale 3 y *aux* pasa a apuntar a `NULL`:



Ahí acaba el bucle. El valor devuelto por la función es 3, el número de nodos de la lista.

Observa que *longitud_lista* tarda más cuanto mayor es la lista. Una lista con *n* nodos obliga a efectuar *n* iteraciones del bucle **for**. Algo similar (aunque sin manejar listas enlazadas) nos ocurría con *strlen*, la función que calcula la longitud de una cadena.

La forma de usar esta función desde el programa principal es sencilla:

miprograma.c
```
#include <stdio.h>
#include "lista.h"

int main(void)
{
  TipoLista lista;
  ...
  printf("Longitud: %d\n", longitud_lista(lista));

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 250 ¿Funcionará correctamente *longitud_lista* cuando le pasamos una lista vacía?

▶ 251 Diseña una función que reciba una lista de enteros con enlace simple y devuelva el valor de su elemento máximo. Si la lista está vacía, se devolverá el valor 0.

▶ 252 Diseña una función que reciba una lista de enteros con enlace simple y devuelva su media. Si la lista está vacía, se devolverá el valor 0.

. . . .

4.6.5. Impresión en pantalla

Ahora que sabemos recorrer una lista no resulta en absoluto difícil diseñar un procedimiento que muestre el contenido de una lista en pantalla. El prototipo es éste:

lista.h
```
...
extern void muestra_lista(TipoLista lista);
```

y una posible implementación, ésta:

lista.c
```
void muestra_lista(TipoLista lista)
{
  struct Nodo * aux;

  for (aux = lista; aux != NULL; aux = aux->sig)
    printf("%d\n", aux->info);
}
```

. . . . EJERCICIOS . . . .

▶ 253 Diseña un procedimiento que muestre el contenido de una lista al estilo Python. Por ejemplo, la lista de la última figura se mostrará como `[3, 8, 2]`. Fíjate en que la coma sólo aparece separando a los diferentes valores, no después de todos los números.

▶ 254 Diseña un procedimiento que muestre el contenido de una lista como se indica en el siguiente ejemplo. La lista formada por los valores 3, 8 y 2 se representará así:

```
->[3]->[8]->[2]->|
```

(La barra vertical representa a `NULL`.)

. . . .

4.6.6. Inserción por cola

Diseñemos ahora una función que inserte un nodo al final de una lista. Su prototipo será:

lista.h

```
...
extern TipoLista inserta_por_cola(TipoLista lista, int valor);
```

Nuestra función se dividirá en dos etapas: una primera que localice al último elemento de la lista, y otra que cree el nuevo nodo y lo una a la lista.

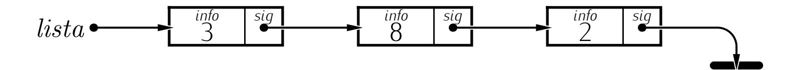
Aquí tienes la primera etapa:

lista.c

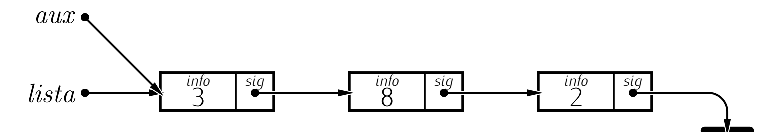
```
TipoLista inserta_por_cola(TipoLista lista, int valor)
{
  struct Nodo * aux;

  for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
  ...
}
```

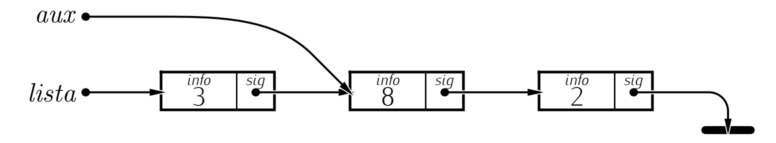
Analicemos paso a paso el bucle con un ejemplo. Imagina que la lista que nos suministran en *lista* ya tiene tres nodos:



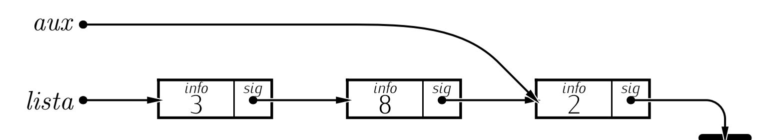
La primera iteración del bucle hace que *aux* apunte al primer elemento de la lista:



Habrá una nueva iteración si *aux*->*sig* es distinto de `NULL`, es decir, si el nodo apuntado por *aux* no es el último de la lista. Es nuestro caso, así que iteramos haciendo *aux* = *aux*->*sig*, o sea, pasamos a esta nueva situación:



¿Sigue siendo cierto que *aux*->*sig* es distinto de `NULL`? Sí. Avanzamos *aux* un nodo más a la derecha:



¿Y ahora? ¿Es cierto que *aux*->*sig* es distinto de NULL? No, es igual a NULL. Ya hemos llegado al último nodo de la lista. Fíjate en que hemos parado un paso antes que cuando contábamos el número de nodos de una lista; entonces la condición de iteración del bucle era otra: «*aux* != NULL».

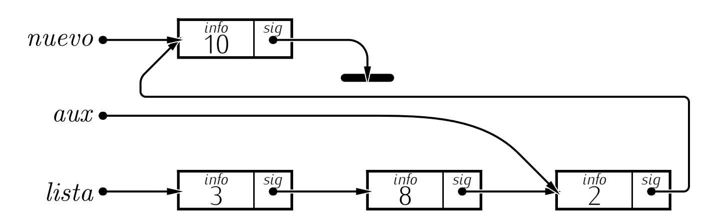
Podemos proceder con la segunda fase de la inserción: pedir un nuevo nodo y enlazarlo desde el actual último nodo. Nos vendrá bien un nuevo puntero auxiliar:

↯ lista.c ↯

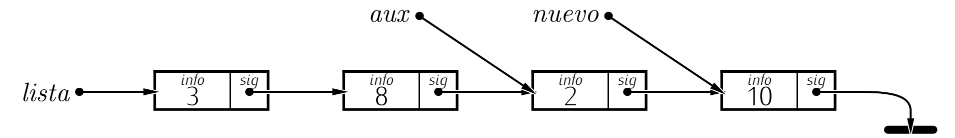
```
TipoLista inserta_por_cola(TipoLista lista, int valor)
{
  struct Nodo * aux, * nuevo;

  for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
  nuevo = malloc(sizeof(struct Nodo));
  nuevo->info = valor;
  nuevo->sig = NULL;
  aux->sig = nuevo;
  return lista;
}
```

El efecto de la ejecución de las nuevas líneas, suponiendo que el valor es 10, es éste:



Está claro que ha funcionado correctamente, ¿no? Tal vez resulte de ayuda ver la misma estructura reordenada así:



Bien, entonces, ¿por qué hemos marcado la función como incorrecta? Veamos qué ocurre si la lista que nos proporcionan está vacía. Si la lista está vacía, *lista* vale NULL. En la primera iteración del bucle **for** asignaremos a *aux* el valor de *lista*, es decir, NULL. Para ver si pasamos a efectuar la primera iteración, hemos de comprobar antes si *aux*->*sig* es distinto de NULL. ¡*Pero es un error preguntar por el valor de aux->sig cuando aux es* NULL! Un puntero a NULL no apunta a nodo alguno, así que no podemos preguntar por el valor del campo *sig* de un nodo que no existe. ¡Ojo con este tipo de errores!: los accesos a memoria que no nos «pertenece» no son detectables por el compilador. Se manifiestan en tiempo de ejecución y, normalmente, con consecuencias desastrosas[6], especialmente al efectuar escrituras de información. ¿Cómo podemos corregir la función? Tratando a la lista vacía como un caso especial:

lista.c

```
TipoLista inserta_por_cola(TipoLista lista, int valor)
{
  struct Nodo * aux, * nuevo;

  if (lista == NULL) {
    lista = malloc(sizeof(struct Nodo));
    lista->info = valor;
```

[6]En Linux, por ejemplo, obtendrás un error (típicamente «Segmentation fault») y se abortará inmediatamente la ejecución del programa. En Microsoft Windows es frecuente que el ordenador «se cuelgue».

```
        lista->sig = NULL;
    }
    else {
        for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
        nuevo = malloc(sizeof(struct Nodo));
        nuevo->info = valor;
        nuevo->sig = NULL;
        aux->sig = nuevo;
    }
    return lista;
}
```

Como puedes ver, el tratamiento de la lista vacía es muy sencillo, pero especial. Ya te lo advertimos antes: comprueba siempre si tu función se comporta adecuadamente en situaciones extremas. La lista vacía es un caso para el que siempre deberías comprobar la validez de tu aproximación.

La función puede retocarse factorizando acciones comunes a los dos bloques del **if-else**:

lista.c

```
TipoLista inserta_por_cola(TipoLista lista, int valor)
{
    struct Nodo * aux, * nuevo;

    nuevo = malloc(sizeof(struct Nodo));
    nuevo->info = valor;
    nuevo->sig = NULL;
    if (lista == NULL)
        lista = nuevo;
    else {
        for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
        aux->sig = nuevo;
    }
    return lista;
}
```

Mejor así.

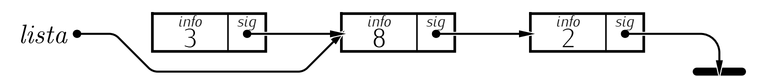
4.6.7. Borrado de la cabeza

La función que vamos a diseñar ahora recibe una lista y devuelve esa misma lista sin el nodo que ocupaba inicialmente la posición de cabeza. El prototipo será éste:

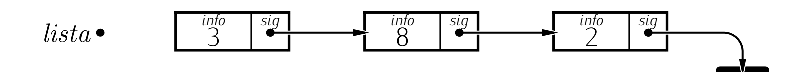
lista.h

```
...
extern TipoLista borra_cabeza(TipoLista lista);
```

Implementémosla. No podemos hacer simplemente *lista* = *lista*->*sig*. Ciertamente, ello conseguiría que, en principio, los nodos que «cuelgan» de *lista* formaran una lista correcta, pero estaríamos provocando una fuga de memoria al no liberar con *free* el nodo de la cabeza (ya lo vimos cuando introdujimos las listas enlazadas):



Tampoco podemos empezar haciendo *free*(*lista*) para liberar el primer nodo, pues entonces perderíamos la referencia al resto de nodos. La memoria quedaría así:



¿Quién apuntaría entonces al primer nodo de la lista?

La solución requiere utilizar un puntero auxiliar:

```
                              ⚡ lista.c ⚡
TipoLista borra_cabeza(TipoLista lista)
{
  struct Nodo * aux;

  aux = lista->sig;
  free(lista);
  lista = aux;
  return lista;
}
```

Ahora sí, ¿no? No. Falla en el caso de que *lista* valga `NULL`, es decir, cuando nos pasan una lista vacía. La asignación *aux* = *lista*->*sig* es errónea si *lista* es `NULL`. Pero la solución es muy sencilla en este caso: si nos piden borrar el nodo de cabeza de una lista vacía, ¿qué hemos de hacer? ¡Absolutamente nada!:

```
                               lista.c
TipoLista borra_cabeza(TipoLista lista)
{
  struct Nodo * aux;

  if (lista != NULL) {
    aux = lista->sig;
    free(lista);
    lista = aux;
  }
  return lista;
}
```

Tenlo siempre presente: si usas la expresión *aux*->*sig* para cualquier puntero *aux*, has de estar completamente seguro de que *aux* no es `NULL`.

4.6.8. Borrado de la cola

Vamos a diseñar ahora una función que elimine el último elemento de una lista. He aquí su prototipo:

```
                               lista.h
...
extern TipoLista borra_cola(TipoLista lista);
```

Nuevamente, dividiremos el trabajo en dos fases:

1. localizar el último nodo de la lista para liberar la memoria que ocupa,
2. y hacer que el hasta ahora penúltimo nodo tenga como valor de su campo *sig* a `NULL`.

La primera fase consistirá básicamente en esto:

```
                              ⚡ lista.c ⚡
TipoLista borra_cola(TipoLista lista)
{
  struct Nodo * aux;
```

```
    for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
    ...
}
```

¡Alto! Este mismo bucle ya nos dió problemas cuando tratamos de insertar por la cola: no funciona correctamente con listas vacías. De todos modos, el problema tiene fácil solución: no tiene sentido borrar nada de una lista vacía.

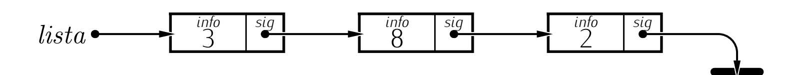
```
lista.c

TipoLista borra_cola(TipoLista lista)
{
    struct Nodo * aux;

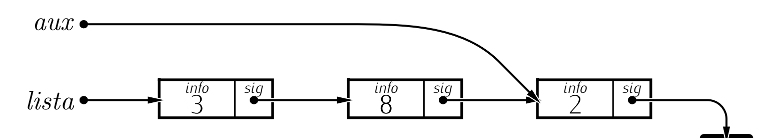
    if (lista != NULL) {
        for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
        ...
    }
    return lista;
}
```

Ahora el bucle solo se ejecuta con listas no vacías. Si partimos de esta lista:

aux



el bucle hace que *aux* acabe apuntando al último nodo:



EJERCICIOS

▶ 255 ¿Seguro que el bucle de *borra_cola* funciona correctamente *siempre*? Piensa si hace lo correcto cuando se le pasa una lista formada por un solo elemento.

Si hemos localizado ya el último nodo de la lista, hemos de liberar su memoria:

```
lista.c

TipoLista borra_cola(TipoLista lista)
{
    struct Nodo * aux;

    if (lista != NULL) {
        for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
        free(aux);
        ...
    }
}
```

Llegamos así a esta situación:

Fíjate: sólo nos falta conseguir que el nuevo último nodo (el de *valor* igual a 8) tenga como valor del campo *sig* a `NULL`. Problema: ¿y cómo sabemos cuál es el último nodo? No se puede saber. Ni siquiera utilizando un nuevo bucle de búsqueda del último nodo, ya que dicho bucle se basaba en que el último nodo es reconocible porque tiene a `NULL` como valor de *sig*, y ahora el último no apunta con *sig* a `NULL`.

El «truco» consiste en usar otro puntero auxiliar y modificar el bucle de búsqueda del último para haga que el nuevo puntero auxiliar vaya siempre «un paso por detrás» de *aux*. Observa:

lista.c

```
TipoLista borra_cola(TipoLista lista)
{
  struct Nodo * aux, * atras;

  if (lista != NULL) {
    for (atras = NULL, aux = lista; aux->sig != NULL; atras = aux, aux = aux->sig) ;
    free(aux);
    ...
  }
}
```

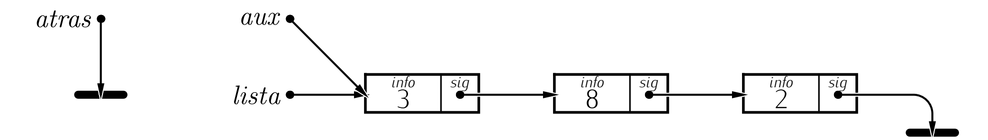
Fíjate en el nuevo aspecto del bucle **for**. Utilizamos una construcción sintáctica que aún no conoces, así que nos detendremos brevemente para explicarla. Los bucles **for** permiten trabajar con más de una inicialización y con más de una acción de paso a la siguiente iteración. Este bucle, por ejemplo, trabaja con dos variables enteras, una que toma valores crecientes y otra que toma valores decrecientes:

```
  for (i=0, j=10; i<3; i++, j--)
    printf("%d %d\n", i, j);
```

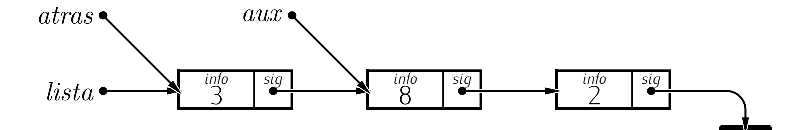
¡Ojo! Es un único bucle, no son dos bucles anidados. ¡No te confundas! Las diferentes inicializaciones y pasos de iteración se separan con comas. Al ejecutarlo, por pantalla aparecerá esto:

```
0 10
1 9
2 8
```

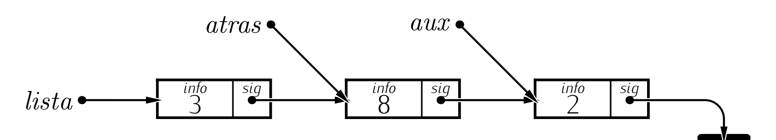
Sigamos con el problema que nos ocupa. Veamos, paso a paso, qué hace ahora el bucle. En la primera iteración tenemos:



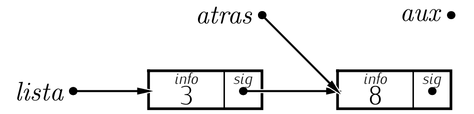
Y en la segunda iteración:



Y en la tercera:



¿Ves? No importa cuán larga sea la lista; el puntero *atras* siempre va un paso por detrás del puntero *aux*. En nuestro ejemplo ya hemos llegado al final de la lista, así que ahora podemos liberar el nodo apuntado por *aux*:

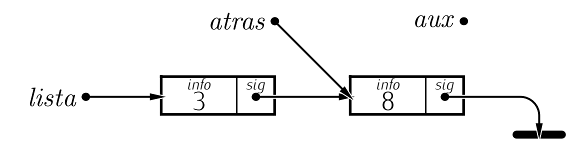


Ahora podemos continuar: ya hemos borrado el último nodo, pero esta vez sí que sabemos cuál es el nuevo último nodo.

```
lista.c
TipoLista borra_cola(TipoLista lista)
{
  struct Nodo * aux, * atras;

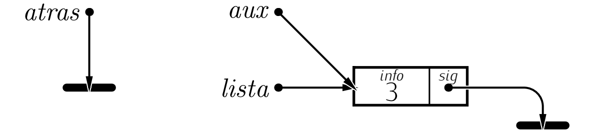
  if (lista != NULL) {
    for (atras = NULL, aux = lista; aux->sig != NULL; atras = aux, aux = aux->sig) ;
    free(aux);
    atras->sig = NULL;
    ...
  }
}
```

Tras ejecutar la nueva sentencia, tenemos:



Aún no hemos acabado. La función *borra_cola* trabaja correctamente con la lista vacía, pues no hace nada en ese caso (no hay nada que borrar), pero, ¿funciona correctamente cuando suministramos una lista con un único elemento? Hagamos una traza.

Tras ejecutar el bucle que busca a los elementos último y penúltimo, los punteros *atras* y *aux* quedan así:



Ahora se libera el nodo apuntado por *aux*:



Y, finalmente, hacemos que *atras*->*sig* sea igual `NULL`. Pero, ¡eso es imposible! El puntero *atras* apunta a `NULL`, y hemos dicho ya que `NULL` no es un nodo y, por tanto, no tiene campo alguno.

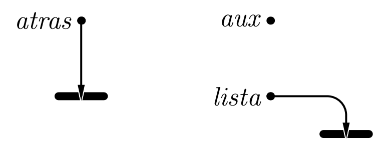
Tratemos este caso como un caso especial. En primer lugar, ¿cómo podemos detectarlo? Viendo si *atras* vale `NULL`. ¿Y qué hemos de hacer entonces? Hemos de hacer que *lista* pase a valer `NULL`, sin más.

```
lista.c
TipoLista borra_cola(TipoLista lista)
{
```

```
  struct Nodo * aux, * atras;

  if (lista != NULL) {
    for (atras = NULL, aux = lista; aux->sig != NULL; atras = aux, aux = aux->sig) ;
    free(aux);
    if (atras == NULL)
      lista = NULL;
    else
      atras->sig = NULL;
  }
  return lista;
}
```

Ya está. Si aplicásemos este nuevo método, nuestro ejemplo concluiría así:



Hemos aprendido una lección: *otro caso especial que conviene estudiar explícitamente es el de la lista compuesta por un solo elemento.*

Insistimos en que debes seguir una sencilla regla en el diseño de funciones con punteros: si accedes a un campo de un puntero *ptr*, por ejemplo, *ptr*->*sig* o *ptr*->*info*, pregúntate siempre si cabe alguna posibilidad de que *ptr* sea `NULL`; si es así, tienes un problema que debes solucionar.

. . . . EJERCICIOS . . . .

▶ 256 ¿Funcionan correctamente las funciones que hemos definido antes (cálculo de la longitud, inserción por cabeza y por cola y borrado de cabeza) cuando se suministra una lista compuesta por un único elemento?

. . . .

4.6.9. Búsqueda de un elemento

Vamos a diseñar ahora una función que no modifica la lista. Se trata de una función que nos indica si un valor entero pertenece a la lista o no. El prototipo de la función será éste:

lista.h

```
...
extern int pertenece(TipoLista lista, int valor);
```

La función devolverá 1 si *valor* está en la lista y 0 en caso contrario.

¿Qué aproximación seguiremos? Pues la misma que seguíamos con los vectores: recorrer cada uno de sus elementos y, si encontramos uno con el valor buscado, devolver inmediatamente el valor 1; si llegamos al final de la lista, será que no lo hemos encontrado, así que en tal caso devolveremos el valor 0.

lista.c

```
int pertenece(TipoLista lista, int valor)
{
  struct Nodo * aux;

  for (aux=lista; aux != NULL; aux = aux->sig)
    if (aux->info == valor)
      return 1;
  return 0;
}
```

Ésta ha sido fácil, ¿no?

. . . . EJERCICIOS . . . .

▶ 257 ¿Funciona correctamente *pertenece* cuando se suministra `NULL` como valor de *lista*, es decir, cuando se suministra una lista vacía? ¿Y cuando se suministra una lista con un único elemento?

. . . .

4.6.10. Borrado del primer nodo con un valor determinado

El problema que abordamos ahora es el diseño de una función que recibe una lista y un valor y elimina el primer nodo de la lista cuyo campo *info* coincide con el valor.

lista.h
```
...
extern TipoLista borra_primera_ocurrencia(TipoLista lista, int valor);
```

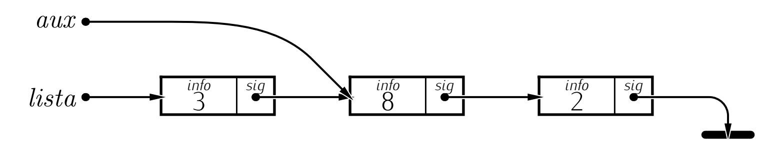
Nuestro primer problema consiste en detectar el valor en la lista. Si el valor no está en la lista, el problema se resuelve de forma trivial: se devuelve la lista intacta y ya está.

ϟ lista.c ϟ
```
TipoLista borra_primera_ocurrencia(TipoLista lista, int valor)
{
  struct Nodo * aux;

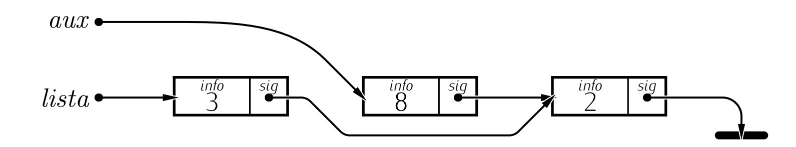
  for (aux=lista; aux != NULL; aux = aux->sig)
    if (aux->info == valor) {
      ...
    }
  return lista;

}
```

Veamos con un ejemplo en qué situación estamos cuando llegamos a la línea marcada con puntos suspensivos. En esta lista hemos buscado el valor 8, así que podemos representar la memoria así:



Nuestro objetivo ahora es, por una parte, efectuar el siguiente «empalme» entre nodos:



y, por otra, eliminar el nodo apuntado por *aux*:

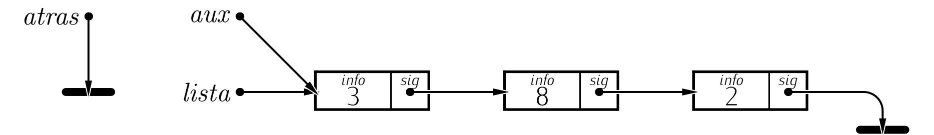
Problema: ¿cómo hacemos el «empalme»? Necesitamos conocer cuál es el nodo que precede al que apunta *aux*. Eso sabemos hacerlo con ayuda de un puntero auxiliar que vaya un paso por detrás de *aux*:

lista.c

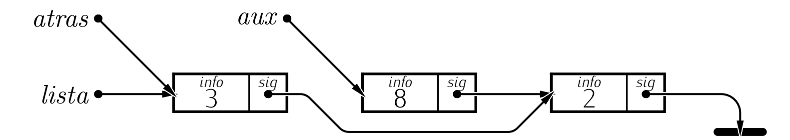
```
TipoLista borra_primera_ocurrencia(TipoLista lista, int valor)
{
  struct Nodo * aux, * atras;

  for (atras = NULL, aux=lista; aux != NULL; atras = aux, aux = aux->sig)
    if (aux->info == valor) {
      atras->sig = aux->sig;
      ...
    }
  return lista;
}
```

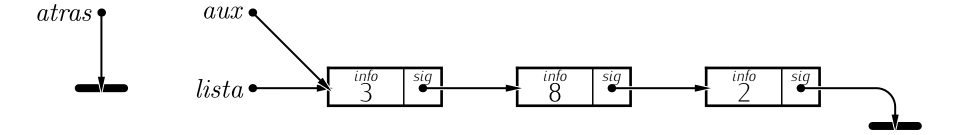
El puntero *atras* empieza apuntando a NULL y siempre va un paso por detrás de *aux*.



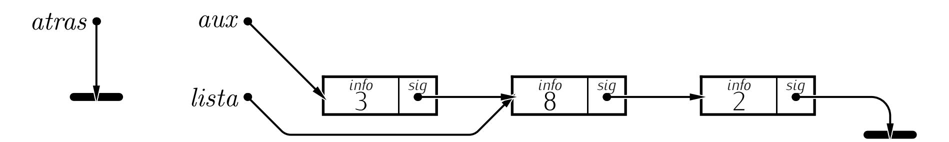
Es decir, cuando *aux* apunta a un nodo, *atras* apunta al anterior. La primera iteración cambia el valor de los punteros y los deja en este estado:



¿Es correcta la función? Hay una fuente de posibles problemas. Estamos asignando algo a *atras->sig*. ¿Cabe alguna posibilidad de que *atras* sea NULL? Sí. El puntero *atras* es NULL cuando el elemento encontrado ocupa la primera posición. Fíjate en este ejemplo en el que queremos borrar el elemento de valor 3:



El «empalme» procedente en este caso es éste:



lista.c

```
TipoLista borra_primera_ocurrencia(TipoLista lista, int valor)
{
  struct Nodo * aux, * atras;

  for (atras = NULL, aux=lista; aux != NULL; atras = aux, aux = aux->sig)
    if (aux->info == valor) {
      if (atras == NULL)
```

```
            lista = aux->sig;
         else
            atras->sig = aux->sig;
         ...
      }
   return lista;
}
```

Ahora podemos borrar el elemento apuntado por *aux* con tranquilidad y devolver la lista modificada:

lista.c

```
TipoLista borra_primera_ocurrencia(TipoLista lista, int valor)
{
   struct Nodo * aux, * atras;

   for (atras = NULL, aux=lista; aux != NULL; atras = aux, aux = aux->sig)
      if (aux->info == valor) {
         if (atras == NULL)
            lista = aux->sig;
         else
            atras->sig = aux->sig;
         free(aux);
         return lista;
      }
   return lista;
}
```

EJERCICIOS

▶ 258 ¿Funciona *borra_primera_ocurrencia* cuando ningún nodo de la lista contiene el valor buscado?

▶ 259 ¿Funciona correctamente en los siguientes casos?

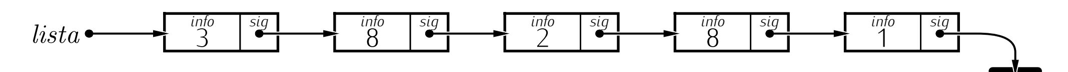
- lista vacía;
- lista con un sólo elemento que no coincide en valor con el buscado;
- lista con un sólo elemento que coincide en valor con el buscado.

Si no es así, corrige la función.

4.6.11. Borrado de todos los nodos con un valor dado

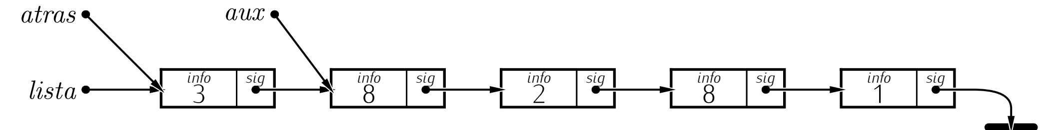
Borrar todos los nodos con un valor dado y no sólo el primero es bastante más complicado, aunque hay una idea que conduce a una solución trivial: llamar tantas veces a la función que hemos diseñado en el apartado anterior como elementos haya originalmente en la lista. Pero, como comprenderás, se trata de una aproximación muy ineficiente: si la lista tiene *n* nodos, llamaremos *n* veces a una función que, en el peor de los casos, recorre la lista completa, es decir, da *n* «pasos» para completarse. Es más eficiente borrar todos los elementos de una sola pasada, en tiempo directamente proporcional a *n*.

Supón que recibimos esta lista:

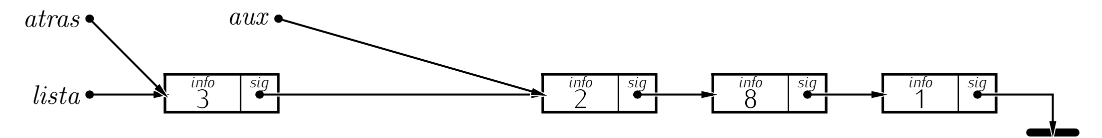


y nos piden eliminar todos los nodos cuyo campo *info* vale 8.

Nuestro problema es localizar el primer 8 y borrarlo dejando los dos punteros auxiliares en un estado tal que podamos seguir iterando para encontrar y borrar el siguiente 8 en la lista (y así con todos los que haya). Ya sabemos cómo localizar el primer 8. Si usamos un bucle con dos punteros (*aux* y *atras*), llegamos a esta situación:



Si eliminamos el nodo apuntado por *aux*, nos interesa que *aux* pase a apuntar al siguiente, pero que *atras* quede apuntando al mismo nodo al que apunta ahora (siempre ha de ir un paso por detrás de *aux*):



Bueno. No resultará tan sencillo. Deberemos tener en cuenta qué ocurre en una situación especial: el borrado del primer elemento de una lista. Aquí tienes una solución:

lista.c

```
TipoLista borra_valor(TipoLista lista, int valor)
{
  struct Nodo * aux, * atras;

  atras = NULL;
  aux = lista;
  while (aux != NULL) {
    if (aux->info == valor) {
      if (atras == NULL)
        lista = aux->sig;
      else
        atras->sig = aux->sig;
      free(aux);
      if (atras == NULL)
        aux = lista;
      else
        aux = atras->sig;
    }
    else {
      atras = aux;
      aux = aux->sig;
    }
  }
  return lista;
}
```

Hemos optado por un bucle **while** en lugar de un bucle **for** porque necesitamos un mayor control de los punteros auxiliares (con el **for**, en cada iteración avanzamos ambos punteros y no siempre queremos que avancen).

. . . . EJERCICIOS . . . .

▶ **260** ¿Funciona *borra_valor* con listas vacías? ¿Y con listas de un sólo elemento? ¿Y con una lista en la que todos los elementos coinciden en valor con el entero que buscamos? Si falla en alguno de estos casos, corrige la función.

. . . .

while y for

Hemos dicho que el bucle **for** no resulta conveniente cuando queremos tener un gran control sobre los punteros auxiliares. No es cierto. El bucle **for** de C permite emular a cualquier bucle **while**. Aquí tienes una versión de *borra_valor* (eliminación de todos los nodos con un valor dado) que usa un bucle **for**:

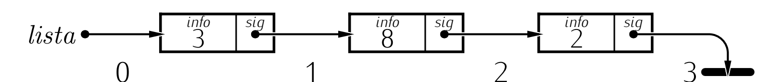
```
TipoLista borra_valor(TipoLista lista, int valor)
{
  struct Nodo * aux, * atras;

  for (atras = NULL, aux = lista; aux != NULL;  ) {
    if (aux->info == valor) {
      if (atras == NULL)
        lista = aux->sig;
      else
        atras->sig = aux->sig;
      free(aux);
      if (atras == NULL)
        aux = lista;
      else
        aux = atras->sig;
    }
    else {
      atras = aux;
      aux = aux->sig;
    }
  }
  return lista;
}
```

Observa que en el bucle **for** hemos dejado en blanco la zona que indica cómo modificar los punteros *aux* y *atras*. Puede hacerse. De hecho, puedes dejar en blanco cualquiera de los componentes de un bucle **for**. Una alternativa a **while** (1), por ejemplo, es **for** (;;).

4.6.12. Inserción en una posición dada

Vamos a diseñar una función que permite insertar un nodo en una posición dada de la lista. Asumiremos la siguiente numeración de posiciones en una lista:



O sea, insertar en la posición 0 es insertar una nueva cabeza; en la posición 1, un nuevo segundo nodo, etc. ¿Qué pasa si se quiere insertar un nodo en una posición mayor que la longitud de la lista? Lo insertaremos en última posición.

El prototipo de la función será:

lista.h
```
...
extern TipoLista inserta_en_posicion(TipoLista lista, int pos, int valor);
```

y aquí tienes su implementación:

lista.c
```
TipoLista inserta_en_posicion(TipoLista lista, int pos, int valor)
```

```
{
   struct Nodo * aux, * atras, * nuevo;
   int i;

   nuevo = malloc(sizeof(struct Nodo));
   nuevo->info = valor;

   for (i=0, atras=NULL, aux=lista; i<pos && aux!=NULL; i++, atras=aux, aux=aux->sig);
   nuevo->sig = aux;
   if (atras == NULL)
      lista = nuevo;
   else
      atras->sig = nuevo;
   return lista;
}
```

EJERCICIOS

▶ 261 Modifica la función para que, si nos pasan un número de posición mayor que el número de elementos de la lista, no se realice inserción alguna.

4.6.13. Inserción ordenada

Las listas que hemos manejado hasta el momento están desordenadas, es decir, sus nodos están dispuestos en un orden arbitrario. Es posible mantener listas ordenadas si las inserciones se realizan utilizando siempre una función que respete el orden.

La función que vamos a desarrollar, por ejemplo, inserta un elemento en una lista ordenada de menor a mayor de modo que la lista resultante sea también una lista ordenada de menor a mayor.

lista.c

```
TipoLista inserta_en_orden(TipoLista lista, int valor);
{
   struct Nodo * aux, * atras, * nuevo;

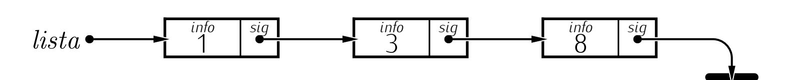
   nuevo = malloc(sizeof(struct Nodo));
   nuevo->info = valor;

   for (atras=NULL, aux=lista; aux != NULL; atras=aux, aux=aux->sig)
      if (valor <= aux->info) {
         /* Aquí insertamos el nodo entre atras y aux. */
         nuevo->sig = aux;
         if (atras == NULL)
            lista = nuevo;
         else
            atras->sig = nuevo;
         /* Y como ya está insertado, acabamos. */
         return lista;
      }
   /* Si llegamos aquí, es que nuevo va al final de la lista. */
   nuevo->sig = NULL;
   if (atras == NULL)
      lista = nuevo;
   else
      atras->sig = nuevo;
   return lista;
}
```

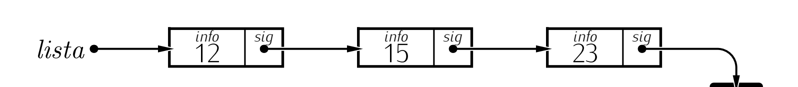
. . . . EJERCICIOS . . . .

▶ 262 Haz una traza de la inserción del valor 7 con *inserta_en_orden* en cada una de estas listas:

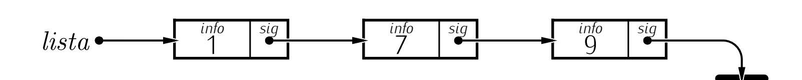
a)



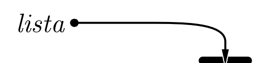
b)



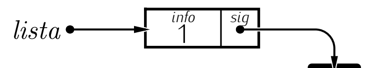
c)



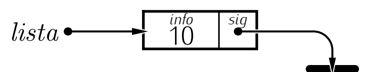
d)



e)



f)



▶ 263 Diseña una función de inserción ordenada en lista que inserte un nuevo nodo si y sólo si no había ningún otro con el mismo valor.

▶ 264 Determinar la pertenencia de un valor a una lista ordenada no requiere que recorras siempre toda la lista. Diseña una función que determine la pertenencia a una lista ordenada efectuando el menor número posible de comparaciones y desplazamientos sobre la lista.

▶ 265 Implementa una función que ordene una lista cualquiera mediante el método de la burbuja.

▶ 266 Diseña una función que diga, devolviendo el valor 1 o el valor 0, si una lista está ordenada o desordenada.

. . . .

4.6.14. Concatenación de dos listas

La función que diseñaremos ahora recibe dos listas y devuelve una nueva lista que resulta de concatenar (una copia de) ambas.

lista.c

```
TipoLista concatena_listas(TipoLista a, TipoLista b)
{
  TipoLista c = NULL;
  struct Nodo * aux, * nuevo, * anterior = NULL;

  for (aux = a; aux != NULL; aux = aux->sig) {
    nuevo = malloc( sizeof(struct Nodo) );
    nuevo->info = aux->info;
    if (anterior != NULL)
      anterior->sig = nuevo;
    else
      c = nuevo;
    anterior = nuevo;
  }
  for (aux = b; aux != NULL; aux = aux->sig) {
    nuevo = malloc( sizeof(struct Nodo) );
    nuevo->info = aux->info;
    if (anterior != NULL)
      anterior->sig = nuevo;
    else
      c = nuevo;
    anterior = nuevo;
  }
  if (anterior != NULL)
    anterior->sig = NULL;
  return c;
}
```

EJERCICIOS

▶ 267 Diseña una función que añada a una lista una copia de otra lista.

▶ 268 Diseña una función que devuelva una lista con los elementos de otra lista que sean mayores que un valor dado.

▶ 269 Diseña una función que devuelva una lista con los elementos comunes a otras dos listas.

▶ 270 Diseña una función que devuelva una lista que es una copia *invertida* de otra lista.

4.6.15. Borrado de la lista completa

Acabaremos este apartado con una rutina que recibe una lista y borra todos y cada uno de sus nodos. A estas alturas no debería resultarte muy difícil de entender:

lista.c

```
TipoLista libera_lista(TipoLista lista)
{
  struct Nodo *aux, *otroaux;

  aux = lista;
```

```
  while (aux != NULL) {
    otroaux = aux->sig;
    free(aux);
    aux = otroaux;
  }
  return NULL;
}
```

Alternativamente podríamos definir la rutina de liberación como un procedimiento:

lista.c

```
void libera_lista(TipoLista * lista)
{
  struct Nodo *aux, *otroaux;

  aux = *lista;
  while (aux != NULL) {
    otroaux = aux->sig;
    free(aux);
    aux = otroaux;
  }
  *lista = NULL;
}
```

De este modo nos aseguramos de que el puntero *lista* fija su valor a `NULL`.

EJERCICIOS

▶ 271 Diseña una función que devuelva un «corte» de la lista. Se proporcionarán como parámetros dos enteros i y j y se devolverá una lista con una copia de los nodos que ocupan las posiciones i a $j-1$, ambas incluídas.

▶ 272 Diseña una función que elimine un «corte» de la lista. Se proporcionarán como parámetros dos enteros i y j y se eliminarán los nodos que ocupan las posiciones i a $j-1$, ambas incluídas.

4.6.16. Juntando las piezas

Te ofrecemos, a modo de resumen, todas las funciones que hemos desarrollado a lo largo de la sección junto con un programa de prueba (faltan, naturalmente, las funciones cuyo desarrollo se propone como ejercicio).

lista.h

lista.h

```
struct Nodo {
  int info;
  struct Nodo * sig;
};

typedef  struct Nodo *  TipoLista;

extern TipoLista lista_vacia(void);
extern int es_lista_vacia(TipoLista lista);
extern TipoLista inserta_por_cabeza(TipoLista lista, int valor);
extern TipoLista inserta_por_cola(TipoLista lista, int valor);
extern TipoLista borra_cabeza(TipoLista lista);
extern TipoLista borra_cola(TipoLista lista);
extern int longitud_lista(TipoLista lista);
extern void muestra_lista(TipoLista lista);
```

```
extern int pertenece(TipoLista lista, int valor);
extern TipoLista borra_primera_ocurrencia(TipoLista lista, int valor);
extern TipoLista borra_valor(TipoLista lista, int valor);
extern TipoLista inserta_en_posicion(TipoLista lista, int pos, int valor);
extern TipoLista inserta_en_orden(TipoLista lista, int valor);
extern TipoLista concatena_listas(TipoLista a, TipoLista b);
extern TipoLista libera_lista(TipoLista lista);
```

lista.c lista.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lista.h"

TipoLista lista_vacia(void)
{
  return NULL;
}

int es_lista_vacia(TipoLista lista)
{
  return lista == NULL;
}

TipoLista inserta_por_cabeza(TipoLista lista, int valor)
{
  struct Nodo * nuevo = malloc(sizeof(struct Nodo));

  nuevo->info = valor;
  nuevo->sig = lista;
  lista = nuevo;
  return lista;
}

TipoLista inserta_por_cola(TipoLista lista, int valor)
{
  struct Nodo * aux, * nuevo;

  nuevo = malloc(sizeof(struct Nodo));
  nuevo->info = valor;
  nuevo->sig = NULL;
  if (lista == NULL)
    lista = nuevo;
  else {
    for (aux = lista; aux->sig != NULL; aux = aux->sig) ;
    aux->sig = nuevo;
  }
  return lista;
}

TipoLista borra_cabeza(TipoLista lista)
{
  struct Nodo * aux;

  if (lista != NULL) {
    aux = lista->sig;
    free(lista);
    lista = aux;
  }
```

```
    return lista;
}

TipoLista borra_cola(TipoLista lista)
{
    struct Nodo * aux, * atras;

    if (lista != NULL) {
        for (atras = NULL, aux = lista; aux->sig != NULL; atras = aux, aux = aux->sig) ;
        free(aux);
        if (atras == NULL)
            lista = NULL;
        else
            atras->sig = NULL;
    }
    return lista;
}

int longitud_lista(TipoLista lista)
{
    struct Nodo * aux;
    int contador = 0;

    for (aux = lista; aux != NULL; aux = aux->sig)
        contador++;
    return contador;
}

void muestra_lista(TipoLista lista)
{ // Como la solución al ejercicio 254, no como lo vimos en el texto.
    struct Nodo * aux;

    printf("->");
    for (aux = lista; aux != NULL; aux = aux->sig)
        printf("[%d]->", aux->info);
    printf("|\n");
}

int pertenece(TipoLista lista, int valor)
{
    struct Nodo * aux;

    for (aux=lista; aux != NULL; aux = aux->sig)
        if (aux->info == valor)
            return 1;
    return 0;
}

TipoLista borra_primera_ocurrencia(TipoLista lista, int valor)
{
    struct Nodo * aux, * atras;

    for (atras = NULL, aux=lista; aux != NULL; atras = aux, aux = aux->sig)
        if (aux->info == valor) {
            if (atras == NULL)
                lista = aux->sig;
            else
                atras->sig = aux->sig;
            free(aux);
```

```
         return lista;
      }
   return lista;
}

TipoLista borra_valor(TipoLista lista, int valor)
{
   struct Nodo * aux, * atras;

   atras = NULL;
   aux = lista;
   while (aux != NULL) {
      if (aux->info == valor) {
         if (atras == NULL)
            lista = aux->sig;
         else
            atras->sig = aux->sig;
         free(aux);
         if (atras == NULL)
            aux = lista;
         else
            aux = atras->sig;
      }
      else {
         atras = aux;
         aux = aux->sig;
      }
   }
   return lista;
}

TipoLista inserta_en_posicion(TipoLista lista, int pos, int valor)
{
   struct Nodo * aux, * atras, * nuevo;
   int i;

   nuevo = malloc(sizeof(struct Nodo));
   nuevo->info = valor;

   for (i=0, atras=NULL, aux=lista; i<pos && aux!=NULL; i++, atras=aux, aux=aux->sig);
   nuevo->sig = aux;
   if (atras == NULL)
      lista = nuevo;
   else
      atras->sig = nuevo;
   return lista;
}

TipoLista inserta_en_orden(TipoLista lista, int valor)
{
   struct Nodo * aux, * atras, * nuevo;

   nuevo = malloc(sizeof(struct Nodo));
   nuevo->info = valor;

   for (atras = NULL, aux = lista; aux != NULL; atras = aux, aux = aux->sig)
      if (valor <= aux->info) {
         /* Aquí insertamos el nodo entre atras y aux. */
         nuevo->sig = aux;
```

```
        if (atras == NULL)
          lista = nuevo;
        else
          atras->sig = nuevo;
        /* Y como ya está insertado, acabamos. */
        return lista;
      }
    /* Si llegamos aquí, es que nuevo va al final de la lista. */
    nuevo->sig = NULL;
    if (atras == NULL)
      lista = nuevo;
    else
      atras->sig = nuevo;
    return lista;
}

TipoLista concatena_listas(TipoLista a, TipoLista b)
{
    TipoLista c = NULL;
    struct Nodo * aux, * nuevo, * anterior = NULL;

    for (aux = a; aux != NULL; aux = aux->sig) {
      nuevo = malloc( sizeof(struct Nodo) );
      nuevo->info = aux->info;
      if (anterior != NULL)
        anterior->sig = nuevo;
      else
        c = nuevo;
      anterior = nuevo;
    }
    for (aux = b; aux != NULL; aux = aux->sig) {
      nuevo = malloc( sizeof(struct Nodo) );
      nuevo->info = aux->info;
      if (anterior != NULL)
        anterior->sig = nuevo;
      else
        c = nuevo;
      anterior = nuevo;
    }
    if (anterior != NULL)
      anterior->sig = NULL;
    return c;
}

TipoLista libera_lista(TipoLista lista)
{
    struct Nodo *aux, *otroaux;

    aux = lista;
    while (aux != NULL) {
      otroaux = aux->sig;
      free(aux);
      aux = otroaux;
    }
    return NULL;
}
```

prueba_lista.c   prueba_lista.c

```c
#include <stdio.h>

#include "lista.h"

int main(void)
{
  TipoLista l, l2, l3;

  printf("Creación de lista\n");
  l = lista_vacia();
  muestra_lista(l);

  printf("¿Es lista vacía?: %d\n", es_lista_vacia(l));

  printf("Inserción por cabeza de 2, 8, 3\n");
  l = inserta_por_cabeza(l, 2);
  l = inserta_por_cabeza(l, 8);
  l = inserta_por_cabeza(l, 3);
  muestra_lista(l);

  printf("Longitud de la lista: %d\n", longitud_lista(l));

  printf("Inserción por cola de 1, 5, 10\n");
  l = inserta_por_cola(l, 1);
  l = inserta_por_cola(l, 5);
  l = inserta_por_cola(l, 10);
  muestra_lista(l);

  printf("Borrado de cabeza\n");
  l = borra_cabeza(l);
  muestra_lista(l);

  printf("Borrado de cola\n");
  l = borra_cola(l);
  muestra_lista(l);

  printf("¿Pertenece 5 a la lista: %d\n", pertenece(l, 5));
  printf("¿Pertenece 7 a la lista: %d\n", pertenece(l, 7));

  printf("Inserción por cola de 1\n");
  l = inserta_por_cola(l, 1);
  muestra_lista(l);

  printf("Borrado de primera ocurrencia de 1\n");
  l = borra_primera_ocurrencia(l, 1);
  muestra_lista(l);

  printf("Nuevo borrado de primera ocurrencia de 1\n");
  l = borra_primera_ocurrencia(l, 1);
  muestra_lista(l);

  printf("Nuevo borrado de primera ocurrencia de 1 (que no está)\n");
  l = borra_primera_ocurrencia(l, 1);
  muestra_lista(l);

  printf("Inserción por cola y por cabeza de 2\n");
  l = inserta_por_cola(l, 2);
  l = inserta_por_cabeza(l, 2);
  muestra_lista(l);
```

```
    printf("Borrado de todas las ocurrencias de 2\n");
    l = borra_valor(l, 2);
    muestra_lista(l);

    printf("Borrado de todas las ocurrencias de 8\n");
    l = borra_valor(l, 8);
    muestra_lista(l);

    printf("Inserción de 1 en posición 0\n");
    l = inserta_en_posicion(l, 0, 1);
    muestra_lista(l);

    printf("Inserción de 10 en posición 2\n");
    l = inserta_en_posicion(l, 2, 10);
    muestra_lista(l);

    printf("Inserción de 3 en posición 1\n");
    l = inserta_en_posicion(l, 1, 3);
    muestra_lista(l);

    printf("Inserción de 4, 0, 20 y 5 en orden\n");
    l = inserta_en_orden(l, 4);
    l = inserta_en_orden(l, 0);
    l = inserta_en_orden(l, 20);
    l = inserta_en_orden(l, 5);
    muestra_lista(l);

    printf("Creación de una nueva lista con los elementos 30, 40, 50\n");
    l2 = lista_vacia();
    l2 = inserta_por_cola(l2, 30);
    l2 = inserta_por_cola(l2, 40);
    l2 = inserta_por_cola(l2, 50);
    muestra_lista(l2);

    printf("Concatenación de las dos listas para formar una nueva\n");
    l3 = concatena_listas(l, l2);
    muestra_lista(l3);

    printf("Liberación de las tres listas\n");
    l = libera_lista(l);
    l2 = libera_lista(l2);
    l3 = libera_lista(l3);
    muestra_lista(l);
    muestra_lista(l2);
    muestra_lista(l3);

    return 0;
}
```

Recuerda que debes compilar estos programas en al menos dos pasos:

```
$ gcc lista.c -c
$ gcc prueba_lista.c lista.o -o prueba_lista
```

Este es el resultado en pantalla de la ejecución de `prueba_lista`:

```
Creación de lista
->|
```

```
¿Es lista vacía?: 1
Inserción por cabeza de 2, 8, 3
->[3]->[8]->[2]->|
Longitud de la lista: 3
Inserción por cola de 1, 5, 10
->[3]->[8]->[2]->[1]->[5]->[10]->|
Borrado de cabeza
->[8]->[2]->[1]->[5]->[10]->|
Borrado de cola
->[8]->[2]->[1]->[5]->|
¿Pertenece 5 a la lista: 1
¿Pertenece 7 a la lista: 0
Inserción por cola de 1
->[8]->[2]->[1]->[5]->[1]->|
Borrado de primera ocurrencia de 1
->[8]->[2]->[5]->[1]->|
Nuevo borrado de primera ocurrencia de 1
->[8]->[2]->[5]->|
Nuevo borrado de primera ocurrencia de 1 (que no está)
->[8]->[2]->[5]->|
Inserción por cola y por cabeza de 2
->[2]->[8]->[2]->[5]->[2]->|
Borrado de todas las ocurrencias de 2
->[8]->[5]->|
Borrado de todas las ocurrencias de 8
->[5]->|
Inserción de 1 en posición 0
->[1]->[5]->|
Inserción de 10 en posición 2
->[1]->[5]->[10]->|
Inserción de 3 en posición 1
->[1]->[3]->[5]->[10]->|
Inserción de 4, 0, 20 y 5 en orden
->[0]->[1]->[3]->[4]->[5]->[5]->[10]->[20]->|
Creación de una nueva lista con los elementos 30, 40, 50
->[30]->[40]->[50]->|
Concatenación de las dos listas para formar una nueva
->[0]->[1]->[3]->[4]->[5]->[5]->[10]->[20]->[30]->[40]->[50]->|
Liberación de las tres listas
->|
->|
->|
```

4.7. Listas simples con punteros a cabeza y cola

Las listas que hemos estudiado hasta el momento son muy rápidas para, por ejemplo, la inserción de elementos por la cabeza. Como la cabeza está permanentemente apuntada por un puntero, basta con pedir memoria para un nuevo nodo y hacer un par de ajustes con punteros:

- hacer que el nodo que sigue al nuevo nodo sea el que era apuntado por el puntero a cabeza,
- y hacer que el puntero a cabeza apunte ahora al nuevo nodo.

No importa cuán larga sea la lista: la inserción por cabeza es siempre igual de rápida. Requiere una cantidad de tiempo constante. Pero la inserción por cola está seriamente

penalizada en comparación con la inserción por cabeza. Como no sabemos dónde está el último elemento, hemos de recorrer la lista completa cada vez que deseamos añadir por la cola. Una forma de eliminar este problema consiste en mantener siempre dos punteros: uno al primer elemento de la lista y otro al último.

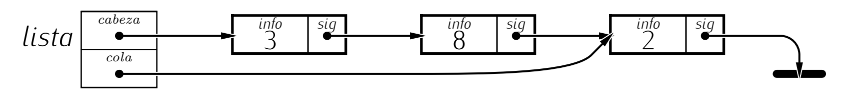
La nueva estructura de datos que representa una lista podría definirse así:

lista_cabeza_cola.h

```
struct Nodo {
   int info;
   struct Nodo * sig;
};

struct Lista_cc {
   struct Nodo * cabeza;
   struct Nodo * cola;
};
```

Podemos representar gráficamente una lista con punteros a cabeza y cola así:



Los punteros *lista.cabeza* y *lista.cola* forman un único objeto del tipo *lista_cc*.

Vamos a presentar ahora unas funciones que gestionan listas con punteros a cabeza y cola. Afortunadamente, todo lo aprendido con las listas del apartado anterior nos vale. Eso sí, algunas operaciones se simplificarán notablemente (añadir por la cola, por ejemplo), pero otras se complicarán ligeramente (eliminar la cola, por ejemplo), ya que ahora hemos de encargarnos de mantener siempre un nuevo puntero (*lista.cola*) apuntando correctamente al último elemento de la lista.

4.7.1. Creación de lista vacía

La función que crea una lista vacía es, nuevamente, muy sencilla. El prototipo es éste:

lista_cabeza_cola.h

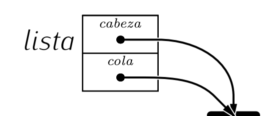
```
extern struct Lista_cc crea_lista_cc_vacia(void);
```

y su implementación:

lista_cabeza_cola.c

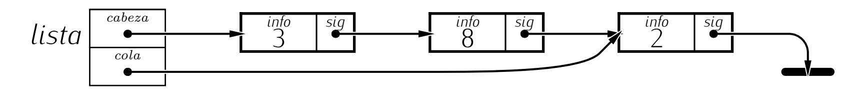
```
struct Lista_cc crea_lista_cc_vacia(void)
{
   struct Lista_cc lista;
   lista.cabeza = lista.cola = NULL;
   return lista;
}
```

Una lista vacía puede representarse así:

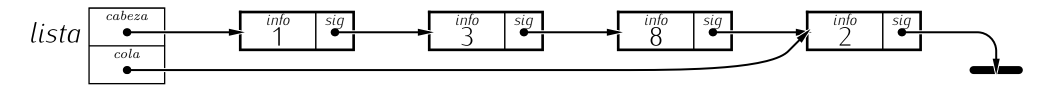


4.7.2. Inserción de nodo en cabeza

La inserción de un nodo en cabeza sólo requiere, en principio, modificar el valor del campo *cabeza*, ¿no? Veamos, si tenemos una lista como ésta:

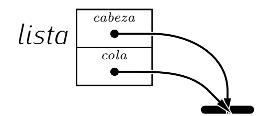


y deseamos insertar el valor 1 en cabeza, basta con modificar *lista.cabeza* y ajustar el campo *sig* del nuevo nodo para que apunte a la antigua cabeza. Como puedes ver, *lista.cola* sigue apuntando al mismo lugar al que apuntaba inicialmente:

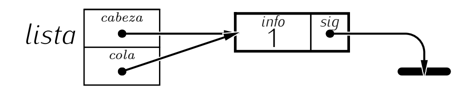


Ya está, ¿no? No. Hay un caso en el que también hemos de modificar *lista.cola* además de *lista.cabeza*: cuando la lista está inicialmente vacía. ¿Por qué? Porque el nuevo nodo de la lista será cabeza y cola a la vez.

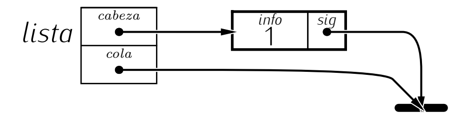
Fíjate, si partimos de esta lista:



e insertamos el valor 1, hemos de construir esta otra:



Si sólo modificásemos el valor de *lista.cabeza*, tendríamos esta otra lista mal formada en la que *lista.cola* no apunta al último elemento:



Ya estamos en condiciones de presentar la función:

lista_cabeza_cola.c

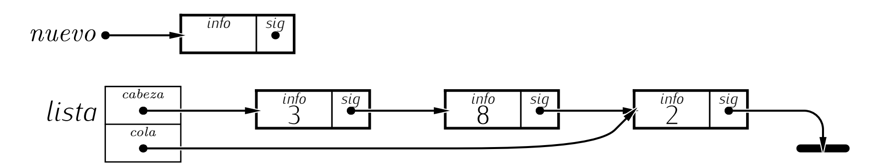
```
struct Lista_cc inserta_por_cabeza(struct Lista_cc lista, int valor)
{
  struct Nodo * nuevo;

  nuevo = malloc(sizeof(struct Nodo));
  nuevo->info = valor;
  nuevo->sig = lista.cabeza;
  if (lista.cabeza == NULL)
    lista.cola = nuevo;
  lista.cabeza = nuevo;
  return lista;
}
```

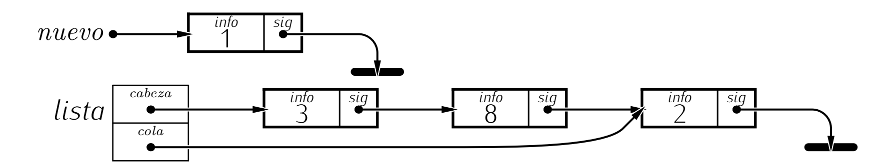
4.7.3. Inserción de nodo en cola

La inserción de un nodo en cola no es mucho más complicada. Como sabemos siempre cuál es el último elemento de la lista, no hemos de buscarlo con un bucle. El procedimiento a seguir es éste:

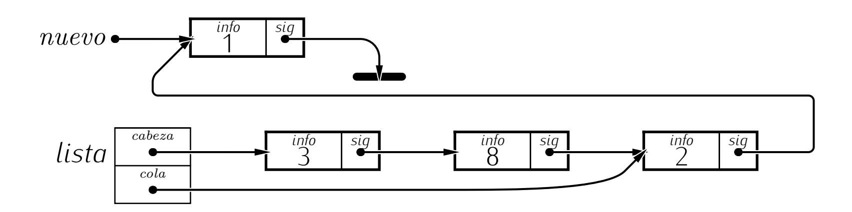
1. Pedimos memoria para un nuevo nodo apuntado por un puntero *nuevo*,



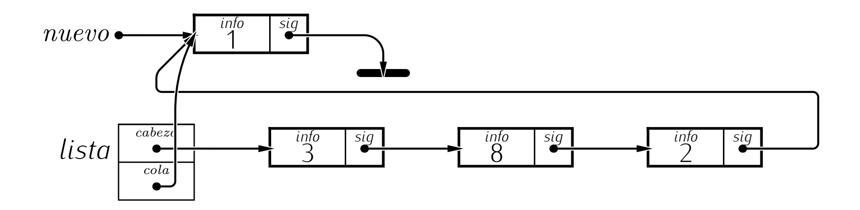
2. asignamos un valor a *nuevo*->*info* y hacemos que el *nuevo*->*sig* sea `NULL`



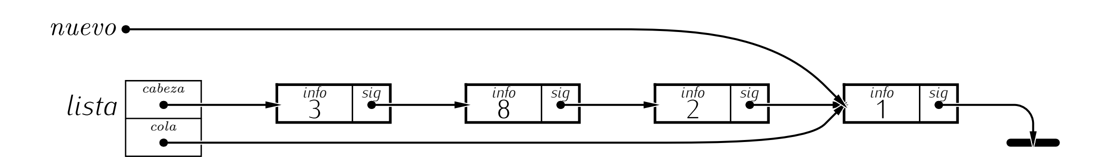
3. hacemos que *lista*.*cola*->*sig* apunte a *nuevo*,



4. y actualizamos *lista*.*cola* para que pase a apuntar a *nuevo*.



Reordenando el gráfico tenemos:



La única precaución que hemos de tener es que, cuando la lista esté inicialmente vacía, se modifique tanto el puntero a la cabeza como el puntero a la cola para que ambos apunten a *nuevo*.

lista_cabeza_cola.c

```
struct Lista_cc inserta_por_cola(struct Lista_cc lista, int valor)
{
  struct Nodo * nuevo;
```

```

  nuevo = malloc(sizeof(struct Nodo));
  nuevo->info = valor;
  nuevo->sig = NULL;

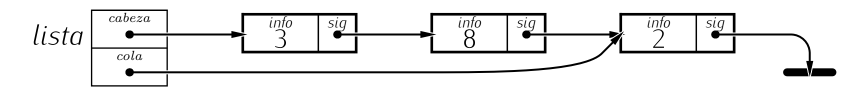
  if (lista.cola != NULL) {
    lista.cola->sig = nuevo;
    lista.cola = nuevo;
  }
  else
    lista.cabeza = lista.cola = nuevo;
  return lista;
}
```

Fíjate: la inserción por cola en este tipo de listas es tan eficiente como la inserción por cabeza. No importa lo larga que sea la lista: siempre cuesta lo mismo insertar por cola, una cantidad constante de tiempo. Acaba de rendir su primer fruto el contar con punteros a cabeza y cola.

4.7.4. Borrado de la cabeza

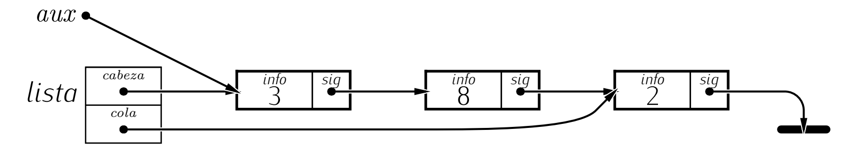
Eliminar un elemento de la cabeza ha de resultar sencillo con la experiencia adquirida:

1. Si la lista está vacía, no hacemos nada.
2. Si la lista tiene un sólo elemento, lo eliminamos y ponemos *lista.cabeza* y *lista.cola* a `NULL`.
3. Y si la lista tiene más de un elemento, como ésta:

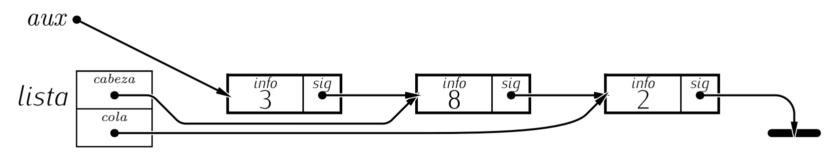


seguimos este proceso:

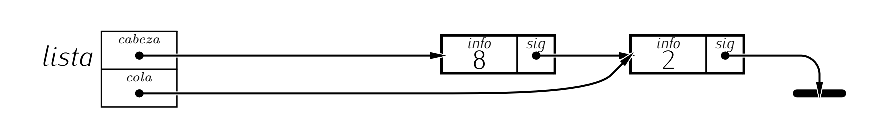
a) Mantenemos un puntero auxiliar apuntando a la actual cabeza,



b) hacemos que *lista.cabeza* apunte al sucesor de la cabeza actual,



c) y liberamos la memoria ocupada por el primer nodo.



lista_cabeza_cola.c

```
struct Lista_cc borra_cabeza(struct Lista_cc lista)
{
   struct Nodo * aux;

   /* Lista vacía: nada que borrar. */
   if (lista.cabeza == NULL)
      return lista;

   /* Lista con un solo nodo: se borra el nodo y la cabeza y la cola pasan a ser NULL. */
   if (lista.cabeza == lista.cola) {
      free(lista.cabeza);
      lista.cabeza = lista.cola = NULL;
      return lista;
   }

   /* Lista con más de un elemento. */
   aux = lista.cabeza;
   lista.cabeza = aux->sig;
   free(aux);
   return lista;
}
```

4.7.5. Borrado de la cola

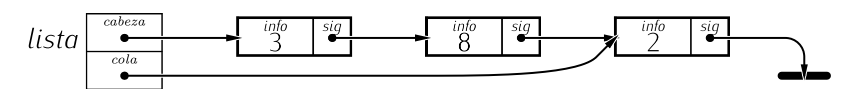
El borrado del último elemento de una lista con punteros a cabeza y cola plantea un problema: cuando hayamos eliminado el nodo apuntado por *lista*.*cola*, ¿a quién debe apuntar *lista*.*cola*? Naturalmente, al que hasta ahora era el penúltimo nodo. ¿Y cómo sabemos cuál era el penúltimo? Sólo hay una forma de saberlo: buscándolo con un recorrido de los nodos de la lista.

Nuevamente distinguiremos tres casos distintos en función de la talla de la lista:

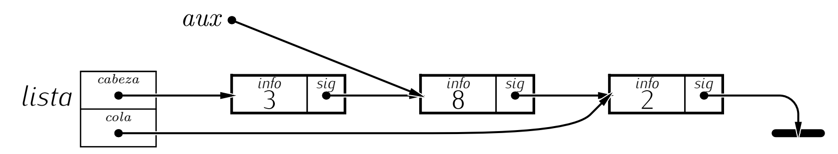
1. Si la lista está vacía, no hacemos nada.

2. Si la lista tiene un único elemento, liberamos su memoria y hacemos que los punteros a cabeza y cola apunten a NULL.

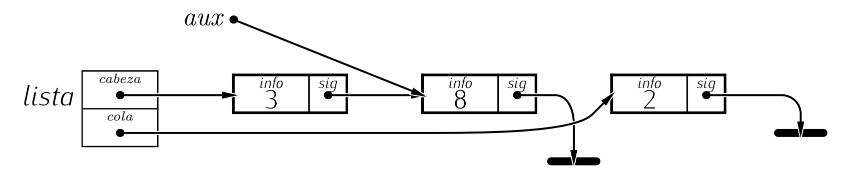
3. En otro caso, actuaremos como en este ejemplo,



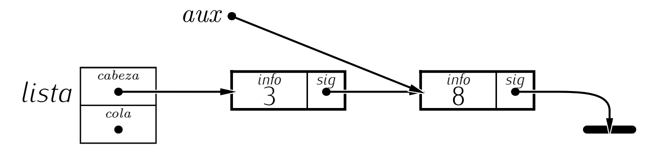
   a) buscamos el *penúltimo* elemento (sabremos cuál es porque si se le apunta con *aux*, entonces *aux*->*sig* coincide con *lista*.*cola*) y lo apuntamos con una variable auxiliar *aux*,



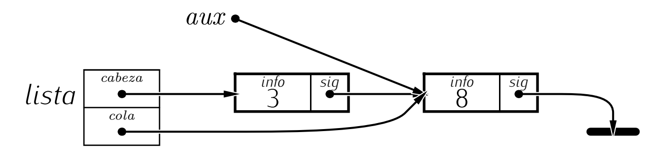
   b) hacemos que el penúltimo no tenga siguiente nodo (ponemos su campo *sig* a NULL) para que así pase a ser el último,



c) liberamos la memoria del que hasta ahora era el último nodo (el apuntado por *lista.cola*)



d) y, finalmente, hacemos que *lista.cola* apunte a *aux*.



lista_cabeza_cola.c

```
struct Lista_cc borra_cola(struct Lista_cc lista)
{
  struct Nodo * aux;

  /* Lista vacía. */
  if (lista.cabeza == NULL)
    return lista;

  /* Lista con un solo nodo. */
  if (lista.cabeza == lista.cola) {
    free(lista.cabeza);
    lista.cabeza = lista.cola = NULL;
    return lista;
  }

  /* Lista con más de un nodo. */
  for (aux = lista.cabeza; aux->sig != lista.cola; aux = aux->sig) ;
  aux->sig = NULL;
  free(lista.cola);
  lista.cola = aux;
  return lista;
}
```

Fíjate en la condición del bucle: detecta si hemos llegado o no al penúltimo nodo preguntando si el que sigue a *aux* es el último (el apuntado por *lista.cola*).

La operación de borrado de la cola no es, pues, tan eficiente como la de borrado de la cabeza, pese a que tenemos un puntero a la cola. El tiempo que necesita es directamente proporcional a la longitud de la lista.

. . . . EJERCICIOS . . . .

▶ 273 Diseña una función que determine si un número pertenece o no a una lista con punteros a cabeza y cola.

▶ 274 Diseña una función que elimine el primer nodo con un valor dado en una lista con punteros a cabeza y cola.

▶ 275 Diseña una función que elimine todos los nodos con un valor dado en una lista con punteros a cabeza y cola.

▶ 276 Diseña una función que devuelva el elemento que ocupa la posición *n* en una lista con puntero a cabeza y cola. (La cabeza ocupa la posición 0.) La función devolverá como valor de retorno 1 o 0 para, respectivamente, indicar si la operación se pudo completar

con éxito o si fracasó. La operación no se puede completar con éxito si n es negativo o si n es mayor o igual que la talla de la lista. El valor del nodo se devolverá en un parámetro pasado por referencia.

▶ 277 Diseña una función que devuelva un «corte» de la lista. Se recibirán dos índices i y j y se devolverá una nueva lista con punteros a cabeza y cola con una copia de los nodos que van del que ocupa la posición i al que ocupa la posición $j-1$, ambos incluídos. La lista devuelta tendrá punteros a cabeza y cola.

▶ 278 Diseña una función de inserción ordenada en una lista ordenada con punteros a cabeza y cola.

▶ 279 Diseña una función que devuelva el menor valor de una lista ordenada con punteros a cabeza y cola.

▶ 280 Diseña una función que devuelva el mayor valor de una lista ordenada con punteros a cabeza y cola.

▶ 281 Diseña una función que añada a una lista con punteros a cabeza y cola una copia de otra lista con punteros a cabeza y cola.

4.8. Listas con enlace doble

Vamos a dotar *a cada nodo* de dos punteros: uno al *siguiente* nodo en la lista y otro al *anterior*.

Los nodos serán variables de este tipo:

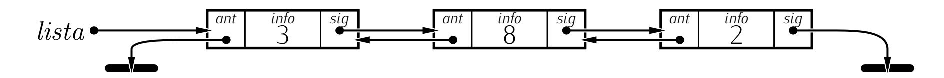
lista_doble.h

```
struct DNodo {
   int info;            // Valor del nodo.
   struct DNodo * ant;  // Puntero al anterior.
   struct DNodo * sig;  // Puntero al siguiente.
};
```

Una lista es un puntero a un **struct** *DNodo* (o a `NULL`). Nuevamente, definiremos un tipo para poner énfasis en que un puntero representa a la lista que «cuelga» de él.

```
typedef struct DNodo * TipoDLista;
```

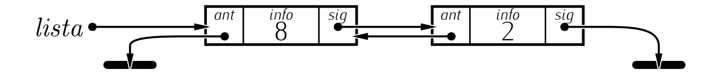
Aquí tienes una representación gráfica de una lista doblemente enlazada:



Observa que cada nodo tiene dos punteros: uno al nodo anterior y otro al siguiente. ¿Qué nodo sigue al último nodo? Ninguno, o sea, `NULL`. ¿Y cuál antecede al primero? Ninguno, es decir, `NULL`.

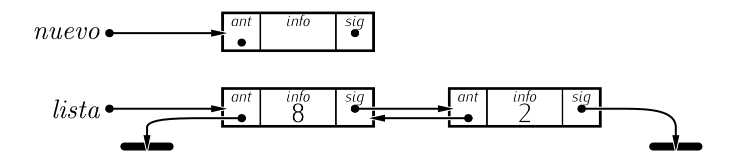
4.8.1. Inserción por cabeza

La inserción por cabeza es relativamente sencilla. Tratemos en primer lugar el caso general: la inserción por cabeza en una lista no vacía. Por ejemplo, en ésta:

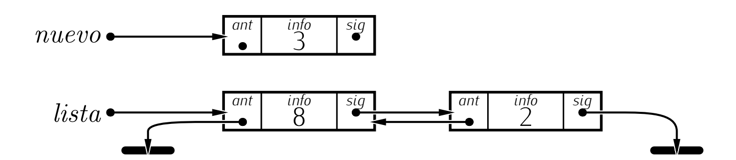


Vamos paso a paso.

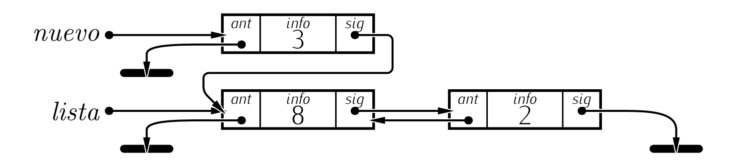
1. Empezamos pidiendo memoria para un nuevo nodo:



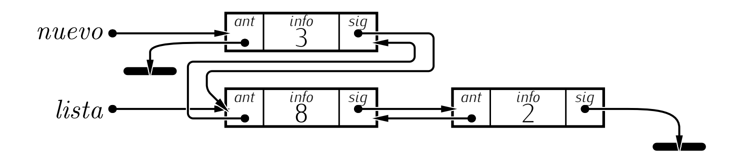
2. Asignamos el valor que nos indiquen al campo *info*:



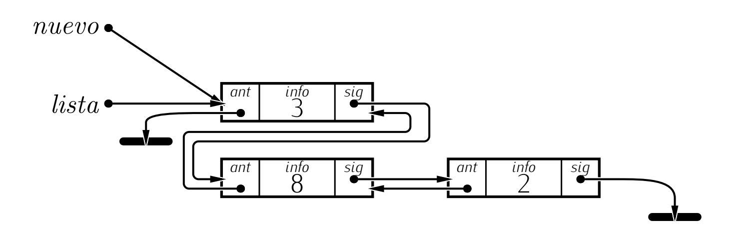
3. Ajustamos sus punteros *ant* y *sig*:



4. Ajustamos el puntero *ant* del que hasta ahora ocupaba la cabeza:



5. Y, finalmente, hacemos que *lista* apunte al nuevo nodo:



El caso de la inserción en la lista vacía es trivial: se pide memoria para un nuevo nodo cuyos punteros *ant* y *sig* se ponen a NULL y hacemos que la cabeza apunte a dicho nodo.

Aquí tienes la función que codifica el método descrito. Hemos factorizado y dispuesto al principio los elementos comunes al caso general y al de la lista vacía:

lista_doble.c

```
TipoDLista inserta_por_cabeza(TipoDLista lista, int valor)
{
  struct DNodo * nuevo;

  nuevo = malloc(sizeof(struct DNodo));
  nuevo->info = valor;
  nuevo->ant = NULL;
  nuevo->sig = lista;

  if (lista != NULL)
```

```
        lista->ant = nuevo;

    lista = nuevo;
    return lista;
}
```

Te proponemos como ejercicios algunas de las funciones básicas para el manejo de listas doblemente enlazadas:

. . . . EJERCICIOS . . . .

▶ 282 Diseña una función que inserte un nuevo nodo al final de una lista doblemente enlazada.

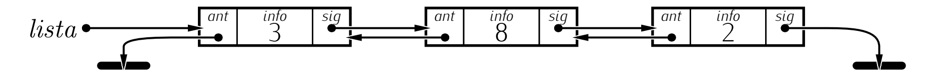
▶ 283 Diseña una función que borre la cabeza de una lista doblemente enlazada. Presta especial atención al caso en el que la lista consta de un sólo elemento.

. . . .

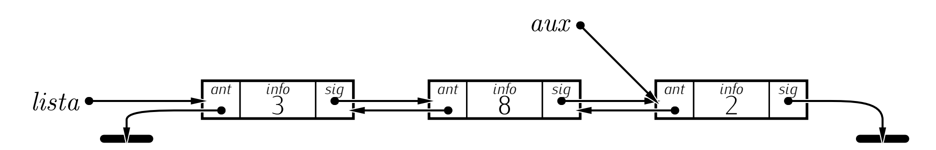
4.8.2. Borrado de la cola

Vamos a desarrollar la función de borrado del último elemento de una lista doblemente enlazada, pues presenta algún aspecto interesante.

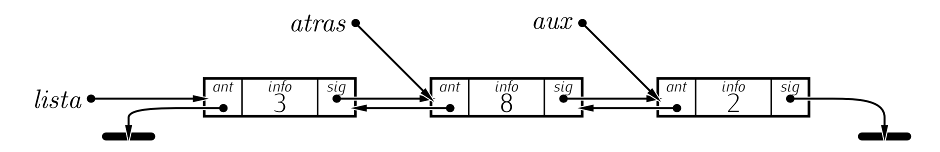
Desarrollemos nuevamente el caso general sobre una lista concreta para deducir el método a seguir. Tomemos, por ejemplo, ésta:



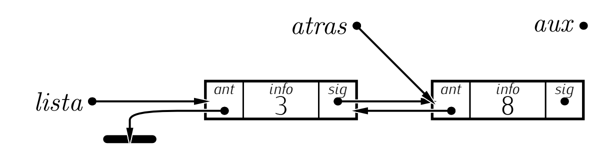
1. Empezamos localizando el *último* elemento de la lista (con un bucle) y apuntándolo con un puntero:



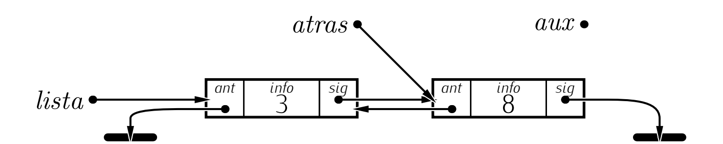
2. Y localizamos ahora el penúltimo en un sólo paso (es *aux*->*ant*):



3. Se elimina el último nodo (el apuntado por *aux*):



4. Y se pone el campo *sig* del que hasta ahora era penúltimo (el apuntado por *atras*) a `NULL`.



El caso de la lista vacía tiene fácil solución: no hay nada que borrar. Es más problemática la lista con sólo un nodo. El problema con ella estriba en que no hay elemento penúltimo (el anterior al último es NULL). Tendremos, pues, que detectar esta situación y tratarla adecuadamente.

lista_doble.c

```
TipoDLista borra_por_cola(TipoDLista lista)
{
   struct DNodo * aux, * atras;

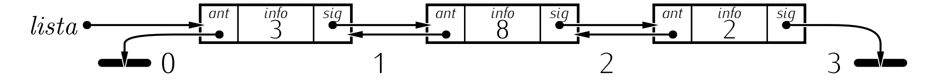
   /* Lista vacía. */
   if (lista == NULL)
      return lista;

/* Lista con un nodo. */
   if (lista->sig == NULL) {
      free(lista);
      lista = NULL;
      return lista;
   }

   /* Caso general. */
   for (aux=lista; aux->sig!=NULL; aux=aux->sig) ;
   atras = aux->ant;
   free(aux);
   atras->sig = NULL;
   return lista;
}
```

4.8.3. Inserción en una posición determinada

Tratemos ahora el caso de la inserción de un nuevo nodo en la posición *n* de una lista doblemente enlazada.

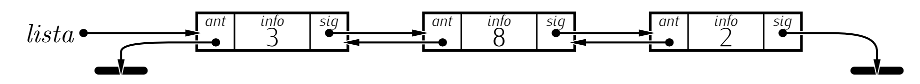


Si *n* está fuera del rango de índices «válidos», insertaremos en la cabeza (si *n* es negativo) o en la cola (si *n* es mayor que el número de elementos de la lista).

A simple vista percibimos ya diferentes casos que requerirán estrategias diferentes:

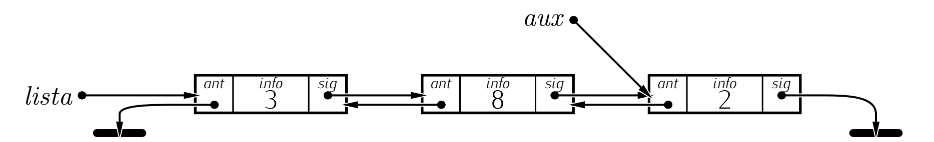
- La lista vacía: la solución en este caso es trivial.
- Inserción al principio de la lista: seguiremos la misma rutina diseñada para insertar por cabeza y, por qué no, utilizaremos la función que diseñamos en su momento.
- Inserción al final de la lista: ídem.[7]
- Inserción entre dos nodos de una lista.

Vamos a desarrollar completamente el último caso. Nuevamente usaremos una lista concreta para deducir cada uno de los detalles del método. Insertaremos el valor 1 en la posición 2 de esta lista:

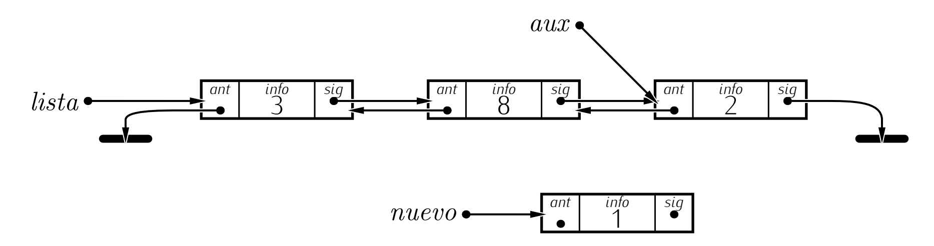


[7]Ver más adelante el ejercicio 285.

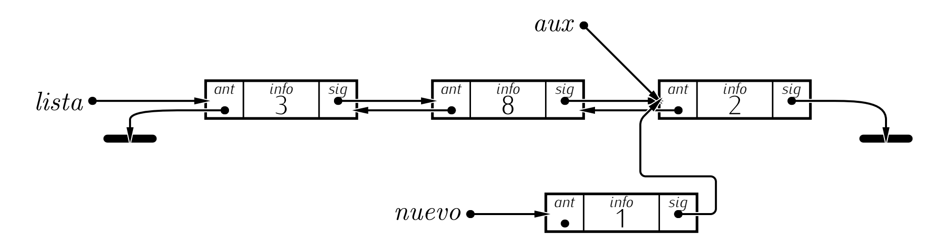
1. Empezamos localizando el elemento que ocupa actualmente la posición *n*. Un simple bucle efectuará esta labor:



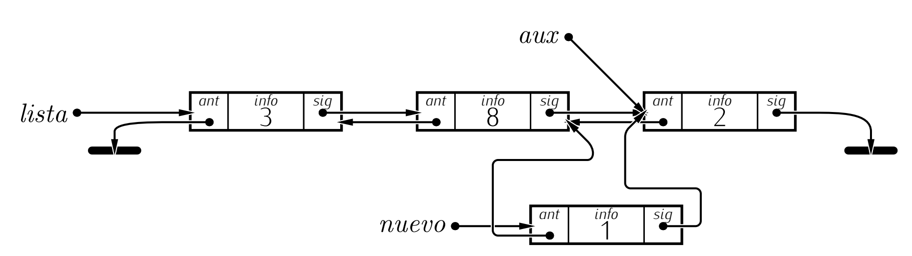
2. Pedimos memoria para un nuevo nodo, lo apuntamos con el puntero *nuevo* y le asignamos el valor:



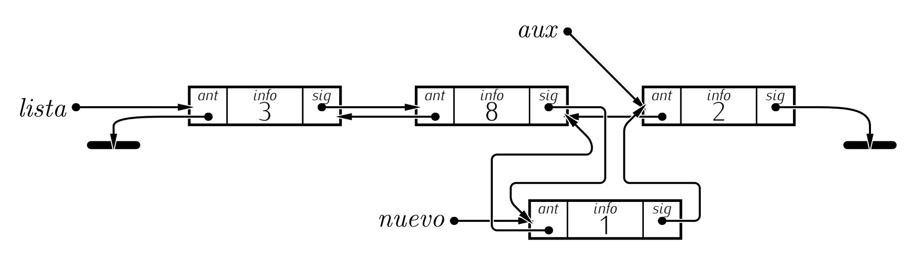
3. Hacemos que *nuevo->sig* sea *aux*:



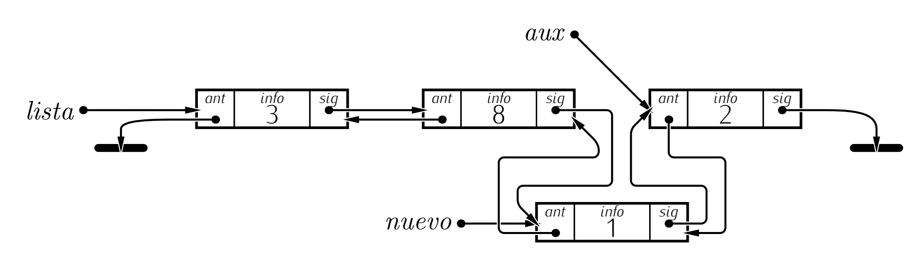
4. Hacemos que *nuevo->ant* sea *aux->ant*:



5. Ojo con este paso, que es complicado. Hacemos que el anterior a *aux* tenga como siguiente a *nuevo*, es decir, *aux->ant->sig = nuevo*:



6. Y ya sólo resta que el anterior a *aux* sea *nuevo* con la asignación *aux->ant = nuevo*:



Ahora que tenemos claro el procedimiento, podemos escribir la función:

lista_doble.c

```
TipoDLista inserta_en_posicion(TipoDLista lista, int pos, int valor)
{
  struct DNodo * aux, * nuevo;
  int i;

  /* Caso especial: lista vacía */
  if (lista == NULL) {
    lista = inserta_por_cabeza(lista, valor);
    return lista;
  }

  /* Inserción en cabeza en lista no vacía. */
  if (pos <= 0) {
    lista = inserta_por_cabeza(lista, valor);
    return lista;
  }

  /* Inserción no en cabeza. */
  nuevo = malloc(sizeof(struct DNodo));
  nuevo->info = valor;
  for (i = 0, aux = lista; i < pos && aux != NULL; i++, aux = aux->sig) ;
  if (aux == NULL) /* Inserción por cola. */
    lista = inserta_por_cola(lista, valor);
  else {
    nuevo->sig = aux;
    nuevo->ant = aux->ant;
    aux->ant->sig = nuevo;
    aux->ant = nuevo;
  }
  return lista;
}
```

EJERCICIOS

▶ 284 Reescribe la función de inserción en una posición dada para que no efectúe llamadas a la función *inserta_por_cabeza*.

▶ 285 Reescribe la función de inserción en una posición dada para que no efectúe llamadas a la función *inserta_por_cola*. ¿Es más eficiente la nueva versión? ¿Por qué?

▶ 286 ¿Qué ocurriría si las últimas líneas de la función fueran éstas?:

```
    ...
    nuevo->sig = aux;
    nuevo->ant = aux->ant;
    aux->ant = nuevo;
    aux->ant->sig = nuevo;
  }
  return lista;
}
```

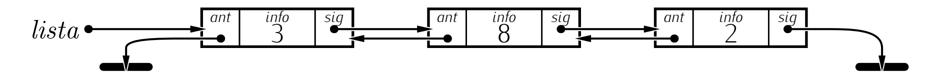
¿Es correcta ahora la función? Haz una traza con un caso concreto.

4.8.4. Borrado de la primera aparición de un elemento

Nuevamente hay un par de casos triviales: si la lista está vacía, no hay que hacer nada y si la lista tiene un sólo elemento, sólo hemos de actuar si ese elemento tiene el valor

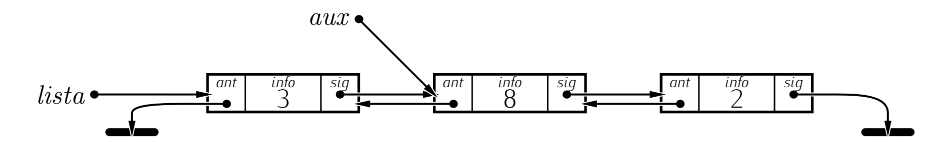
buscado, en cuyo caso liberaremos la memoria del nodo en cuestión y convertiremos la lista en una lista vacía.

Desarrollemos un caso general. Supongamos que en esta lista hemos de eliminar el primer y único nodo con valor 8:

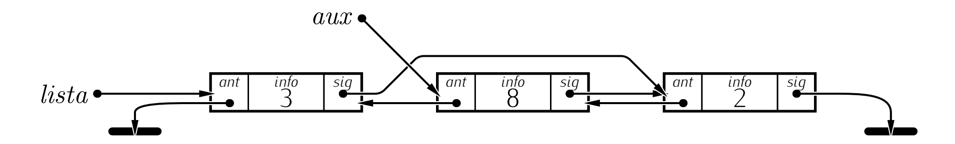


Vamos paso a paso:

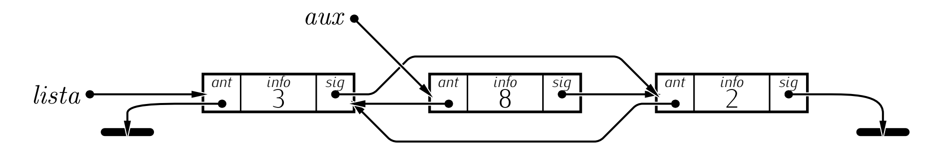
1. Empezamos por localizar el elemento y apuntarlo con un puntero auxiliar *aux*:



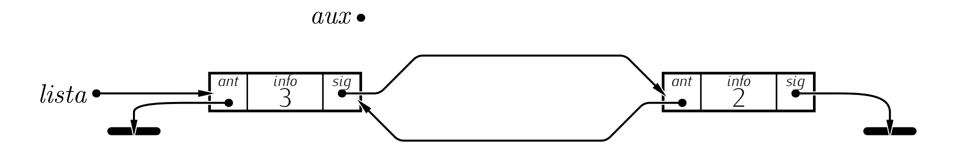
2. Hacemos que el que sigue al anterior de *aux* sea el siguiente de *aux* (¡qué galimatías!). O sea, hacemos *aux*->*ant*->*sig*=*aux*->*sig*:



3. Ahora hacemos que el que antecede al siguiente de *aux* sea el anterior a *aux*. Es decir, *aux*->*sig*->*ant*=*aux*->*ant*:

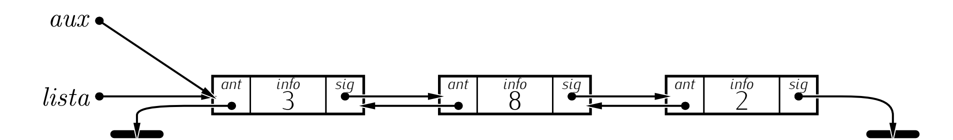


4. Y ya podemos liberar la memoria ocupada por el nodo apuntado con *aux*:

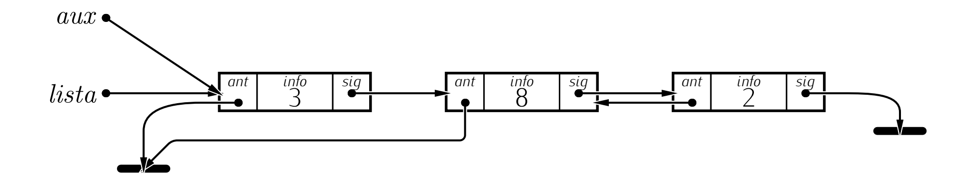


Hemos de ser cautos. Hay un par de casos especiales que merecen ser tratados aparte: el borrado del primer nodo y el borrado del último nodo. Veamos cómo proceder en el primer caso: tratemos de borrar el nodo de valor 3 en la lista del ejemplo anterior.

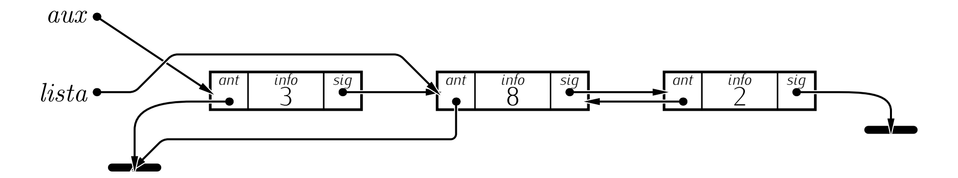
1. Una vez apuntado el nodo por *aux*, sabemos que es el primero porque apunta al mismo nodo que *lista*:



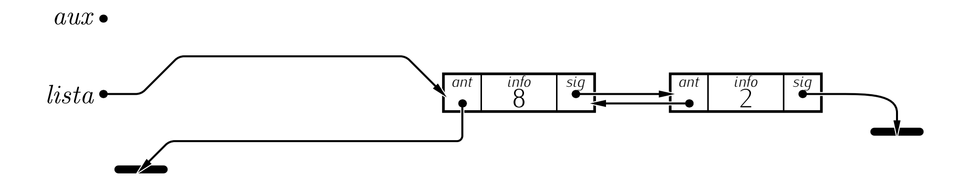
2. Hacemos que el segundo nodo deje de tener antecesor, es decir, que el puntero *aux*->*sig*->*ant* valga `NULL` (que, por otra parte, es lo mismo que hacer *aux*->*sig*->*ant*=*aux*->*ant*):



3. Ahora hacemos que *lista* pase a apuntar al segundo nodo (*lista*=*aux*->*sig*):

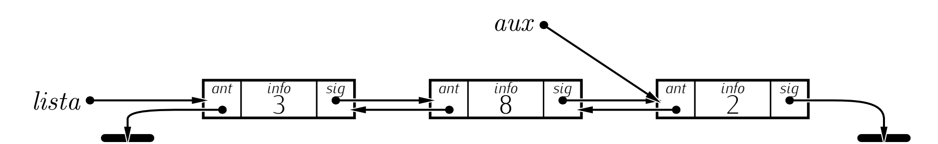


4. Y por fin, podemos liberar al nodo apuntado por *aux* (*free*(*aux*)):

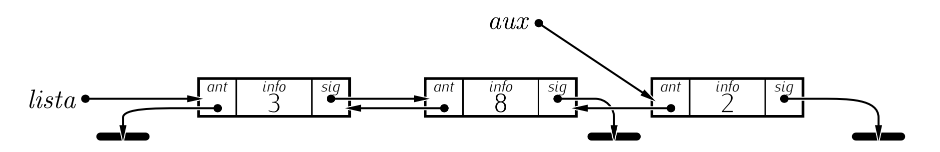


Vamos a por el caso en que borramos el último elemento de la lista:

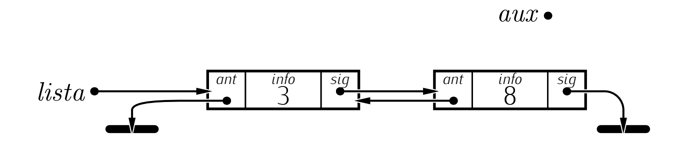
1. Empezamos por localizarlo con *aux* y detectamos que efectivamente es el último porque *aux*->*sig* es `NULL`:



2. Hacemos que el siguiente del que antecede a *aux* sea `NULL`: (*aux*->*ant*->*sig*=`NULL`):



3. Y liberamos el nodo apuntado por *aux*:



lista_doble.c

```
TipoDLista borra_primera_ocurrencia(TipoDLista lista, int valor)
{
  struct DNodo * aux;

  for (aux=lista; aux!=NULL; aux=aux->sig)
    if (aux->info == valor)
      break;

```

```
  if (aux == NULL) // No se encontró.
    return lista;

  if (aux->ant == NULL) // Es el primero de la lista.
    lista = aux->sig;
  else
    aux->ant->sig = aux->sig;

  if (aux->sig != NULL) // No es el último de la lista.
    aux->sig->ant = aux->ant;

  free(aux);

  return lista;
}
```

EJERCICIOS

▶ **287** Diseña una función que permita efectuar la inserción ordenada de un elemento en una lista con enlace doble que está ordenada.

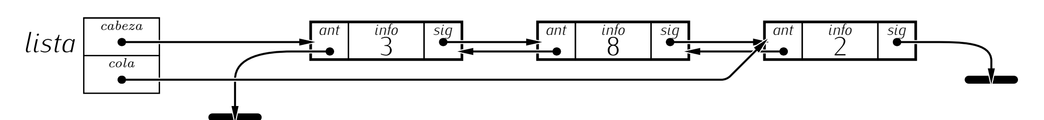
▶ **288** Diseña una función que permita concatenar dos listas doblemente enlazadas. La función recibirá las dos listas y devolverá una lista nueva con una copia de la primera seguida de una copia de la segunda.

▶ **289** Diseña una función que devuelva una copia *invertida* de una lista doblemente enlazada.

4.9. Listas con enlace doble y puntero a cabeza y cola

Ya sabemos manejar listas con puntero a cabeza y listas con punteros a cabeza y cola. Hemos visto que las listas con puntero a cabeza son ineficientes a la hora de añadir elementos por la cola: se tarda tanto más cuanto mayor es el número de elementos de la lista. Las listas con puntero a cabeza y cola permiten realizar operaciones de inserción por cola en un número constante de pasos. Aún así, hay operaciones de cola que también son ineficientes en esta última estructura de datos: la eliminación del nodo de cola, por ejemplo, sigue necesitando un tiempo proporcional a la longitud de la lista.

La estructura que presentamos en esta sección, la lista doblemente enlazada con puntero a cabeza y cola, corrige la ineficiencia en el borrado del nodo de cola. Una lista doblemente enlazada con puntero a cabeza y cola puede representarse gráficamente así:



La definición del tipo es fácil ahora que ya hemos estudiado diferentes tipos de listas:

lista_doble_cc.h

```
struct DNodo {
  int info;
  struct DNodo * ant;
  struct DNodo * sig;
};

struct DLista_cc {
```

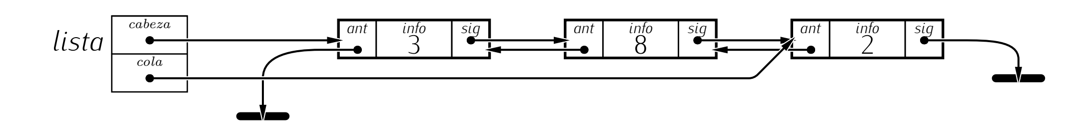
```
    struct DNodo * cabeza;
    struct DNodo * cola;
}

typedef struct DLista_cc TipoDListaCC;
```

Sólo vamos a presentarte una de las operaciones sobre este tipo de listas: el borrado de la cola. El resto de operaciones te las proponemos como ejercicios.

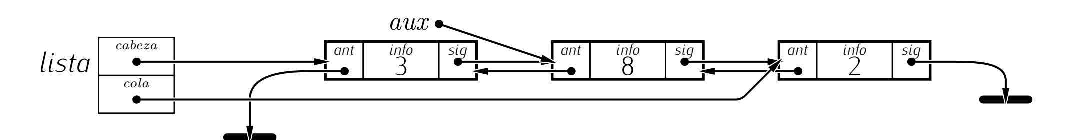
Con cualquiera de las otras estructuras de datos basadas en registros enlazados, el borrado del nodo de cola no podía efectuarse en tiempo constante. Ésta lo hace posible. ¿Cómo? Lo mejor es que, una vez más, despleguemos los diferentes casos y estudiemos ejemplos concretos cuando convenga:

- Si la lista está vacía, no hay que hacer nada.
- Si la lista tiene un solo elemento, liberamos su memoria y ponemos los punteros a cabeza y cola a NULL.
- Y si la lista tiene más de un elemento, como ésta:

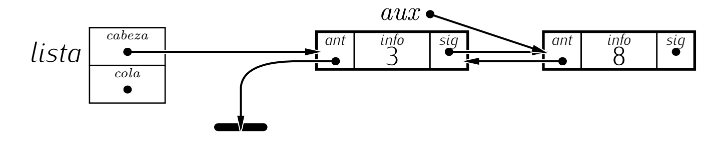


hacemos lo siguiente:

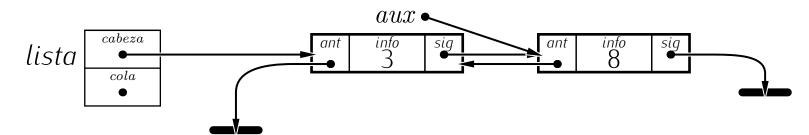
a) localizamos al penúltimo elemento, que es *lista.cola->ant*, y lo mantenemos apuntado con un puntero auxiliar *aux*:



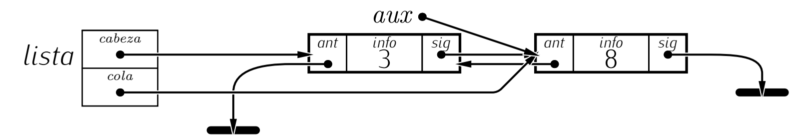
b) liberamos la memoria apuntada por *lista.cola*:



c) ponemos *aux->sig* a NULL:



d) y ajustamos *lista.cola* para que apunte ahora donde apunta *aux*:



Ya podemos escribir el programa:

lista_doble_cc.c

```
TipoDListaCC borra_cola(TipoDListaCC lista)
{
  if (lista.cabeza == NULL)
    return lista;

  if (lista.cabeza == lista.cola) {
    free(lista.cabeza);
    lista.cabeza = lista.cola = NULL;
    return lista;
  }

  aux = lista.cola->ant;
  free(lista.cola);
  aux->sig = NULL;
  lista.cola = aux;
  return lista;
}
```

Ha sido fácil, ¿no? No ha hecho falta bucle alguno. La operación se ejecuta en un número de pasos que es independiente de lo larga que sea la lista.

Ahora te toca a tí desarrollar código. Practica con estos ejercicios:

EJERCICIOS

▶ 290 Diseña una función que calcule la longitud de una lista doblemente enlazada con punteros a cabeza y cola.

▶ 291 Diseña una función que permita insertar un nuevo nodo en cabeza.

▶ 292 Diseña una función que permita insertar un nuevo nodo en cola.

▶ 293 Diseña una función que permita borrar el nodo de cabeza.

▶ 294 Diseña una función que elimine el primer elemento de la lista con un valor dado.

▶ 295 Diseña una función que elimine todos los elementos de la lista con un valor dado.

▶ 296 Diseña una función que inserte un nodo en una posición determinada que se indica por su índice.

▶ 297 Diseña una función que inserte ordenadamente en una lista ordenada.

▶ 298 Diseña una función que muestre por pantalla el contenido de una lista, mostrando el valor de cada celda en una línea. Los elementos se mostrarán en el mismo orden con el que aparecen en la lista.

▶ 299 Diseña una función que muestre por pantalla el contenido de una lista, mostrando el valor de cada celda en un línea. Los elementos se mostrarán en orden inverso.

▶ 300 Diseña una función que devuelva una copia *invertida* de una lista doblemente enlazada con puntero a cabeza y cola.

4.10. Una guía para elegir listas

Hemos estudiado cuatro tipos diferentes de listas basadas en registros enlazados. ¿Por qué tantas? Porque cada una supone una solución de compromiso diferente entre velocidad y consumo de memoria.

Empecemos por estudiar el consumo de memoria. Supongamos que una variable del tipo del campo *info* ocupa *m* bytes, que cada puntero ocupa 4 bytes y que la lista consta

de n elementos. Esta tabla muestra la ocupación en bytes según la estructura de datos escogida: lista con enlace simple («simple»), lista con enlace simple y puntero a cabeza y cola («simple cabeza/cola»), lista con enlace doble («doble»), lista con enlace doble y puntero a cabeza y cola («doble cabeza/cola»).

	memoria (bytes)
simple	$4 + n \cdot (4 + m)$
simple cabeza/cola	$8 + n \cdot (4 + m)$
doble	$4 + n \cdot (8 + m)$
doble cabeza/cola	$8 + n \cdot (8 + m)$

Esta otra tabla resume el tiempo que requieren algunas operaciones sobre los cuatro tipos de lista:

	simple	**simple cabeza/cola**	**doble**	**doble cabeza/cola**
Insertar por cabeza	constante	constante	constante	constante
Borrar cabeza	constante	constante	constante	constante
Insertar por cola	lineal	constante	lineal	constante
Borrar cola	lineal	lineal	lineal	constante
Buscar un nodo concreto	lineal*	lineal*	lineal*	lineal*
Invertir la lista	cuadrático	cuadrático	lineal	lineal

Hemos indicado con la palabra «constante» que se requiere una cantidad de tiempo fija, independiente de la longitud de la lista; con la palabra «lineal», que se requiere un tiempo que es proporcional a la longitud de la lista; y con «cuadrático», que el coste crece con el cuadrado del número de elementos.

Para que te hagas una idea: insertar por cabeza un nodo en una lista cuesta siempre la misma cantidad de tiempo, tenga la lista 100 o 1000 nodos. Insertar por la cola en una lista simplemente enlazada con puntero a cabeza, sin embargo, es unas 10 veces más lento si la lista es 10 veces más larga. Esto no ocurre con una lista simplemente enlazada que tenga puntero a cabeza y cola: insertar por la cola en ella siempre cuesta lo mismo. ¡Ojo con los costes cuadráticos! Invertir una lista simplemente enlazada de 1000 elementos es *100* veces más costoso que invertir una lista con *10* veces menos elementos.

En la tabla hemos marcado algunos costes con un asterisco. Son costes para el peor de los casos. Buscar un nodo concreto en una lista obliga a recorrer todos los nodos sólo si el que buscamos no está o si ocupa la última posición. En el mejor de los casos, el coste temporal es constante: ello ocurre cuando el nodo buscado se encuentra en la lista y, además, ocupa la primera posición. De los análisis de coste nos ocuparemos más adelante.

Un análisis de la tabla de tiempos permite concluir que la lista doblemente enlazada con punteros a cabeza y cola es siempre igual o mejor que las otras estructuras. ¿Debemos escogerla siempre? No, por tres razones:

1. Aunque la lista más compleja requiere tiempo constante en muchas operaciones, éstas son algo más lentas y sofisticadas que operaciones análogas en las otras estructuras más sencillas. Son, por fuerza, algo más lentas.

2. El consumo de memoria es mayor en la lista más compleja (8 bytes adicionales para cada nodo y 8 bytes para los punteros a cabeza y cola, frente a 4 bytes adicionales para cada nodo y 4 bytes para un puntero a cabeza en la estructura más sencilla), así que puede no compensar la ganancia en velocidad o, sencillamente, es posible que no podamos permitirnos el lujo de gastar el doble de memoria extra.

3. Puede que nuestra aplicación sólo efectúe operaciones «baratas» sobre cualquier lista. Imagina que necesitas una lista en la que siempre insertas y eliminas nodos por cabeza, jamás por el final. Las cuatro estructuras ofrecen tiempo constante para

esas dos operaciones, sólo que, además, las dos primeras son mucho más sencillas y consumen menos memoria que las dos últimas.

. . . . EJERCICIOS . . . .

▶ 301 Rellena una tabla similar a la anterior para estas otras operaciones:

a) Insertar ordenadamente en una lista ordenada.

b) Insertar en una posición concreta.

c) Buscar un elemento en una lista ordenada.

d) Buscar el elemento de valor mínimo en una lista ordenada.

e) Buscar el elemento de valor máximo en una lista ordenada.

f) Unir dos listas ordenadas de modo que el resultado esté ordenado.

g) Mostrar el contenido de una lista por pantalla.

h) Mostrar el contenido de una lista en orden inverso por pantalla.

▶ 302 Vamos a montar una pila con listas. La pila es una estructura de datos en la que sólo podemos efectuar las siguientes operaciones:

- insertar un elemento en la cima,
- eliminar el elemento de la cima,
- consultar el valor del elemento de la cima.

¿Qué tipo de lista te parece más adecuado para implementar una pila? ¿Por qué?

▶ 303 Vamos a montar una cola con listas. La cola es una estructura de datos en la que sólo podemos efectuar las siguientes operaciones:

- insertar un elemento al final de la cola,
- eliminar el elemento que hay al principio de la cola,
- consultar el valor del elemento que hay al principio de la cola.

¿Qué tipo de lista te parece más adecuado para construir una cola? ¿Por qué?

. . . .

4.11. Una aplicación: una base de datos para discos compactos

En este apartado vamos a desarrollar una aplicación práctica que usa listas: un programa para la gestión de una colección de discos compactos. Cada disco compacto contendrá un título, un intérprete, un año de edición y una lista de canciones. De cada canción nos interesará únicamente el título.

Las acciones del programa, que se presentarán al usuario con un menú, son éstas.

1. Añadir un disco.
2. Buscar discos por título.
3. Buscar discos por intérprete.
4. Buscar discos por título de canción.

5. Mostrar el contenido completo de la colección.

6. Eliminar un disco de la base de datos dados su título y el nombre del intérprete.

7. Finalizar.

A priori no sabemos cuántas canciones hay en un disco, ni cuántos discos hay que almacenar en la base de datos, así que utilizaremos listas para ambas entidades. Nuestra colección será, pues, una lista de discos que, a su vez, contienen listas de canciones. No sólo eso: no queremos que nuestra aplicación desperdicie memoria con cadenas que consumen más memoria que la necesaria, así que usaremos memoria dinámica también para la reserva de memoria para cadenas.

Lo mejor es dividir el problema en estructuras de datos claramente diferenciadas (una para la lista de discos y otra para la lista de canciones) y diseñar funciones para manejar cada una de ellas. Atención al montaje que vamos a presentar, pues es el más complicado de cuantos hemos estudiado.

```
struct Cancion {
   char * titulo;
   struct Cancion * sig;
};

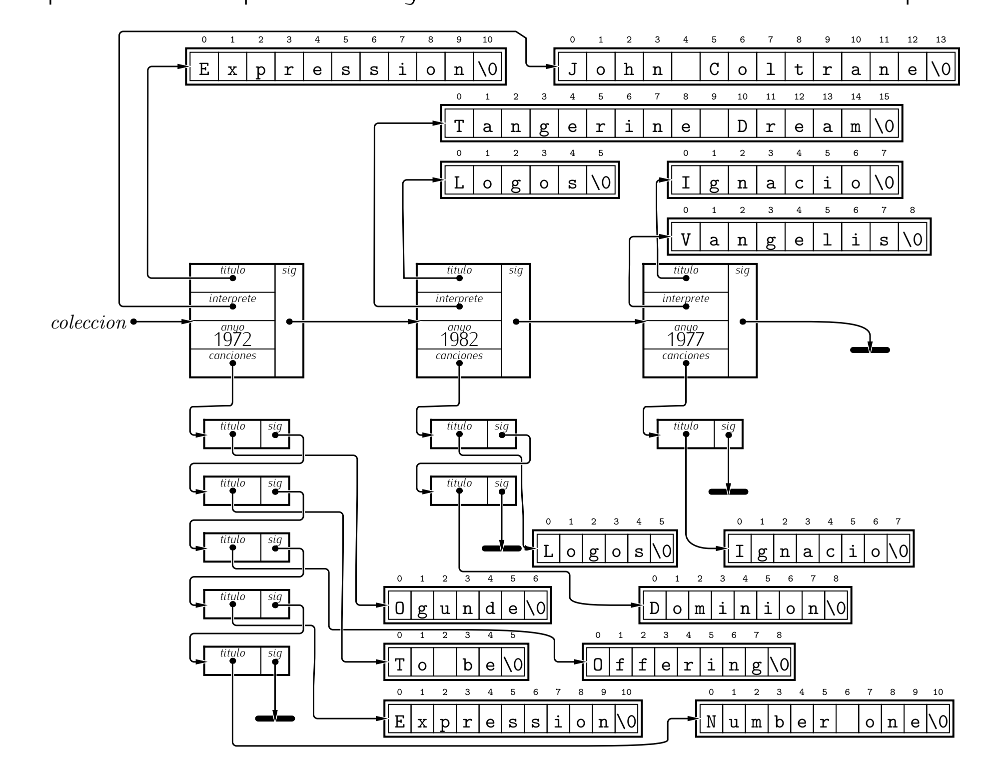
typedef struct Cancion * TipoListaCanciones;

struct Disco {
   char * titulo;
   char * interprete;
   int anyo;
   TipoListaCanciones canciones;
   struct Disco * sig;
};

typedef struct Disco * TipoColeccion;
```

Hemos optado por listas simplemente enlazadas y con puntero a cabeza.

Aquí tienes una representación gráfica de una colección con 3 discos compactos:



Empezaremos por diseñar la estructura que corresponde a una lista de canciones. Después nos ocuparemos del diseño de registros del tipo «disco compacto». Y acabaremos definiendo un tipo «colección de discos compactos».

Vamos a diseñar funciones para gestionar listas de canciones. Lo que no vamos a hacer es montar *toda* posible operación sobre una lista. Sólo invertiremos esfuerzo en las operaciones que se van a utilizar. Éstas son:

- Crear una lista vacía.
- Añadir una canción a la lista. (Durante la creación de un disco iremos pidiendo las canciones y añadiéndolas a la ficha del disco.)
- Mostrar la lista de canciones por pantalla. (Esta función se usará cuando se muestre una ficha detallada de un disco.)
- Buscar una canción y decir si está o no está en la lista.
- Borrar todas las canciones de la lista. (Cuando se elimine un disco de la base de datos tendremos que liberar la memoria ocupada por todas sus canciones.)

La función de creación de una lista de canciones es trivial:

```
TipoListaCanciones crea_lista_canciones(void)
{
  return NULL;
}
```

Pasemos a la función que añade una canción a una lista de canciones. No nos indican que las canciones deban almacenarse en un orden determinado, así que recurriremos al método más sencillo: la inserción por cabeza.

```
TipoListaCanciones anyade_cancion(TipoListaCanciones lista, char titulo[])
{
  struct Cancion * nuevo = malloc(sizeof(struct Cancion));

  nuevo->titulo = malloc((strlen(titulo)+1)*sizeof(char));
  strcpy(nuevo->titulo, titulo);
  nuevo->sig = lista;
  lista = nuevo;
  return lista;
}
```

. . . . EJERCICIOS . . . .

▶ 304 La verdad es que insertar las canciones por la cabeza es el método menos indicado, pues cuando se recorra la lista para mostrarlas por pantalla aparecerán en orden inverso a aquél con el que fueron introducidas. Modifica *anyade_cancion* para que las canciones se inserten por la cola.

▶ 305 Y ya que sugerimos que insertes canciones por cola, modifica las estructuras necesarias para que la lista de canciones se gestione con una lista de registros con puntero a cabeza y cola.

. . . .

Mostrar la lista de canciones es muy sencillo:

```
void muestra_canciones(TipoListaCanciones lista)
{
  struct Cancion * aux;

  for (aux=lista; aux!=NULL; aux=aux->sig)
    printf("  %s\n", aux->titulo);
}
```

Buscar una canción es un simple recorrido que puede terminar anticipadamente tan pronto se encuentra el objeto buscado:

```
int contiene_cancion_con_titulo(TipoListaCanciones lista, char titulo[])
{
  struct Cancion * aux;

  for (aux=lista; aux!=NULL; aux=aux->sig)
    if (strcmp(aux->titulo, titulo)==0)
      return 1;
  return 0;
}
```

Borrar todas las canciones de una lista debe liberar la memoria propia de cada nodo, pero también debe liberar la cadena que almacena cada título, pues también se solicitó con *malloc*:

```
TipoListaCanciones libera_canciones(TipoListaCanciones lista)
{
  struct Cancion * aux, * siguiente;

  aux = lista;
  while (aux != NULL) {
    siguiente = aux->sig;
    free(aux->titulo);
    free(aux);
    aux = siguiente;
  }
  return NULL;
}
```

No ha sido tan difícil. Una vez sabemos manejar listas, las aplicaciones prácticas se diseñan reutilizando buena parte de las rutinas que hemos presentado en apartados anteriores.

Pasamos a encargarnos de las funciones que gestionan la lista de discos. Como es habitual, empezamos con una función que crea una colección (una lista) vacía:

```
TipoColeccion crea_coleccion(void)
{
  return NULL;
}
```

Añadir un disco obliga a solicitar memoria tanto para el registro en sí como para algunos de sus componentes: el título y el intérprete:

```
TipoColeccion anyade_disco(TipoColeccion lista, char titulo[], char interprete[],
                           int anyo, TipoListaCanciones canciones)
{
  struct Disco * disco;

  disco = malloc(sizeof(struct Disco));
  disco->titulo = malloc((strlen(titulo)+1)*sizeof(char));
  strcpy(disco->titulo, titulo);
  disco->interprete = malloc((strlen(interprete)+1)*sizeof(char));
  strcpy(disco->interprete, interprete);
  disco->anyo = anyo;
  disco->canciones = canciones;
  disco->sig = lista;
  lista = disco;
  return lista;
}
```

. . . . EJERCICIOS . . . .

▶ **306** Modifica *anyade_disco* para que los discos estén siempre ordenados alfabéticamente por intérprete y, para cada intérprete, por valor creciente del año de edición.

. . . .

Y la memoria solicitada debe liberarse íntegramente: si al reservar memoria para un disco ejecutamos tres llamadas a *malloc*, habrá que efectuar tres llamadas a *free*:

```
TipoColeccion libera_coleccion(TipoColeccion lista)
{
  struct Disco * aux, * siguiente;

  aux = lista;
  while (aux != NULL) {
    siguiente = aux->sig;
    free(aux->titulo);
    free(aux->interprete);
    aux->canciones = libera_canciones(aux->canciones);
    free(aux);
    aux = siguiente;
  }
  return NULL;
}
```

Mostrar por pantalla el contenido de un disco es sencillo, especialmente si usamos *muestra_canciones* para mostrar la lista de canciones.

```
void muestra_disco(struct Disco eldisco)
{
  printf("Título:␣%s\n", eldisco.titulo);
  printf("Intérprete:␣%s\n", eldisco.interprete);
  printf("Año␣de␣edición:␣%d\n", eldisco.anyo);
  printf("Canciones:\n");
  muestra_canciones(eldisco.canciones);
}
```

Mostrar la colección completa es trivial si usamos la función que muestra un disco:

```
void muestra_coleccion(TipoColeccion lista)
{
  struct Disco * aux;

  for (aux=lista; aux!=NULL; aux=aux->sig)
    muestra_disco(*aux);
}
```

Las funciones de búsqueda de discos se usan en un contexto determinado: el de mostrar, si se encuentra el disco, su contenido por pantalla. En lugar de hacer que la función devuelva el valor 1 o 0, podemos hacer que devuelva un puntero al registro cuando lo encuentre o NULL cuando el disco no esté en la base de datos. Aquí tienes las funciones de búsqueda por título y por intérprete:

```
struct Disco * busca_disco_por_titulo_disco(TipoColeccion coleccion, char titulo[])
{
  struct Disco * aux;

  for (aux=coleccion; aux!=NULL; aux=aux->sig)
    if (strcmp(aux->titulo, titulo) == 0)
      return aux;
  return NULL;
}

```

```
struct Disco * busca_disco_por_interprete(TipoColeccion coleccion, char interprete[])
{
  struct Disco * aux;

  for (aux=coleccion; aux!=NULL; aux=aux->sig)
    if (strcmp(aux->interprete, interprete) == 0)
      return aux;
  return NULL;
}
```

La función de búsqueda por título de canción es similar, sólo que llama a la función que busca una canción en una lista de canciones:

```
struct Disco * busca_disco_por_titulo_cancion(TipoColeccion coleccion, char titulo[])
{
  struct Disco * aux;

  for (aux=coleccion; aux!=NULL; aux=aux->sig)
    if (contiene_cancion_con_titulo(aux->canciones, titulo))
      return aux;
  return NULL;
}
```

Sólo nos queda por definir la función que elimina un disco de la colección dado su título:

```
TipoColeccion borra_disco_por_titulo_e_interprete(TipoColeccion coleccion, char titulo[],
                                                   char interprete[])
{
  struct Disco *aux, *atras;

  for (atras = NULL, aux=coleccion; aux != NULL; atras = aux, aux = aux->sig)
    if (strcmp(aux->titulo, titulo) == 0 && strcmp(aux->interprete, interprete) == 0) {
      if (atras == NULL)
        coleccion = aux->sig;
      else
        atras->sig = aux->sig;
      free(aux->titulo);
      free(aux->interprete);
      aux->canciones = libera_canciones(aux->canciones);
      free(aux);
      return coleccion;
    }
  return coleccion;
}
```

Ya tenemos todas las herramientas para enfrentarnos al programa principal:

discoteca2.c discoteca2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXCAD 1000

enum { Anyadir=1, BuscarPorTituloDisco, BuscarPorInterprete, BuscarPorTituloCancion,
       Mostrar, EliminarDisco, Salir};

  ⋮
```

```
int main(void)
{
   int opcion;
   TipoColeccion coleccion;
   char titulo_disco[MAXCAD+1], titulo_cancion[MAXCAD+1], interprete[MAXCAD+1];
   char linea[MAXCAD+1];
   int anyo;
   struct Disco * undisco;
   TipoListaCanciones lista_canciones;

   coleccion = crea_coleccion();

   do {
      printf("Menú\n");
      printf("1) Añadir disco\n");
      printf("2) Buscar por título del disco\n");
      printf("3) Buscar por intérprete\n");
      printf("4) Buscar por título de canción\n");
      printf("5) Mostrar todo\n");
      printf("6) Eliminar un disco por título e intérprte\n");
      printf("7) Finalizar\n");
      printf("Opción: "); gets(linea); sscanf(linea, "%d", &opcion);

      switch(opcion) {
         case Anyadir:
            printf("Título: "); gets(titulo_disco);
            printf("Intérprete: "); gets(interprete);
            printf("Año: "); gets(linea); sscanf(linea, "%d", &anyo);
            lista_canciones = crea_lista_canciones();
            do {
               printf("Título de canción (pulse retorno para acabar):  ");
               gets(titulo_cancion);
               if (strlen(titulo_cancion) > 0)
                  lista_canciones = anyade_cancion(lista_canciones, titulo_cancion);
            } while (strlen(titulo_cancion) > 0);
            coleccion = anyade_disco(coleccion, titulo_disco,
                              interprete, anyo, lista_canciones);
            break;

         case BuscarPorTituloDisco:
            printf("Título: "); gets(titulo_disco);
            undisco = busca_disco_por_titulo_disco(coleccion, titulo_disco);
            if (undisco != NULL)
               muestra_disco(*undisco);
            else
               printf("No hay discos con título '%s'\n", titulo_disco);
            break;

         case BuscarPorInterprete:
            printf("Intérprete: "); gets(interprete);
            undisco = busca_disco_por_interprete(coleccion, interprete);
            if (undisco != NULL)
               muestra_disco(*undisco);
            else
               printf("No hay discos de %s\n", interprete);
            break;

         case BuscarPorTituloCancion:
```

```
        printf("Título: "); gets(titulo_cancion);
        undisco = busca_disco_por_titulo_cancion(coleccion, titulo_cancion);
        if (undisco != NULL)
          muestra_disco(*undisco);
        else
          printf("No hay discos con alguna canción titulada '%s'\n",
                 titulo_cancion);
        break;

      case Mostrar:
        muestra_coleccion(coleccion);
        break;

      case EliminarDisco:
        printf("Título: "); gets(titulo_disco);
        printf("Intérprete: "); gets(interprete);
        coleccion = borra_disco_por_titulo_e_interprete(coleccion, titulo_disco,
                 interprete);
        break;
    }
  } while (opcion != Salir);

  coleccion = libera_coleccion(coleccion);

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ **307** Modifica el programa para que se almacene la duración de cada canción (en segundos) junto al título de la misma.

▶ **308** La función de búsqueda de discos por intérprete se detiene al encontrar el primer disco de un intérprete dado. Modifica la función para que devuelva una lista con una copia de todos los discos de un intérprete. Usa esa lista para mostrar su contenido por pantalla con *muestra_coleccion* y elimínala una vez hayas mostrado su contenido.

▶ **309** Diseña una aplicación para la gestión de libros de una biblioteca. Debes mantener dos listas: una lista de libros y otra de socios. De cada socio recordamos el nombre, el DNI y el teléfono. De cada libro mantenemos los siguientes datos: título, autor, ISBN, código de la biblioteca (una cadena con 10 caracteres) y estado. El estado es un puntero que, cuando vale `NULL`, indica que el libro está disponible y, en caso contrario, apunta al socio al que se ha prestado el libro.

El programa debe permitir dar de alta y baja libros y socios, así como efectuar el préstamo de un libro a un socio y gestionar su devolución. Ten en cuenta que no es posible dar de baja a un socio que posee un libro en préstamo ni dar de baja un libro prestado.

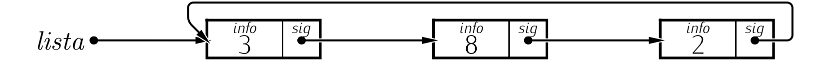
. . . .

4.12. Otras estructuras de datos con registros enlazados

La posibilidad de trabajar con registros enlazados abre las puertas al diseño de estructuras de datos muy elaboradas que permiten efectuar ciertas operaciones muy eficientemente. El precio a pagar es una mayor complejidad de nuestros programas C y, posiblemente, un mayor consumo de memoria (estamos almacenando valores *y* punteros, aunque sólo nos interesan los valores).

Pero no has visto más que el principio. En otras asignaturas de la carrera aprenderás a utilizar estructuras de datos complejas, pero capaces de ofrecer tiempos de respuesta mucho mejores que las listas que hemos estudiado o capaces de permitir implementaciones sencillas para operaciones que aún no hemos estudiado. Te vamos a presentar unos pocos ejemplos ilustrativos.

- Las *listas circulares*, por ejemplo, son listas sin final. El nodo siguiente al que parece el último nodo es el primero. Ningún nodo está ligado a `NULL`.

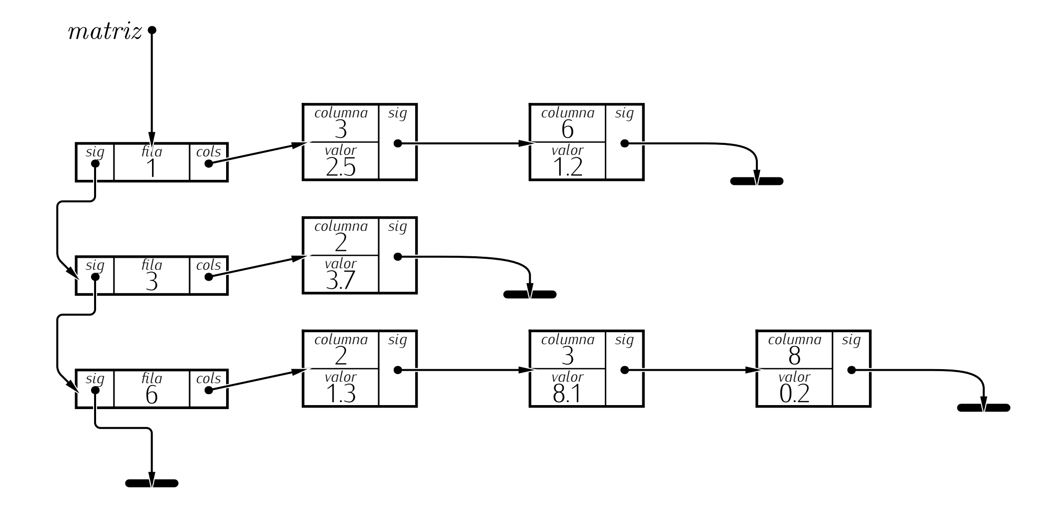
  

  Este tipo de estructura de datos es útil, por ejemplo, para mantener una lista de tareas a las que hay que ir dedicando atención rotativamente: cuando hemos hecho una ronda, queremos pasar nuevamente al primer elemento. El campo *sig* del último elemento permite pasar directamente al primero, con lo que resulta sencillo codificar un bucle que recorre rotativamente la lista.

- En muchas aplicaciones es preciso trabajar con *matrices dispersas*. Una matriz dispersa es una matriz en la que muy pocos componentes presentan un valor diferente de cero. Esta matriz, por ejemplo, es dispersa:

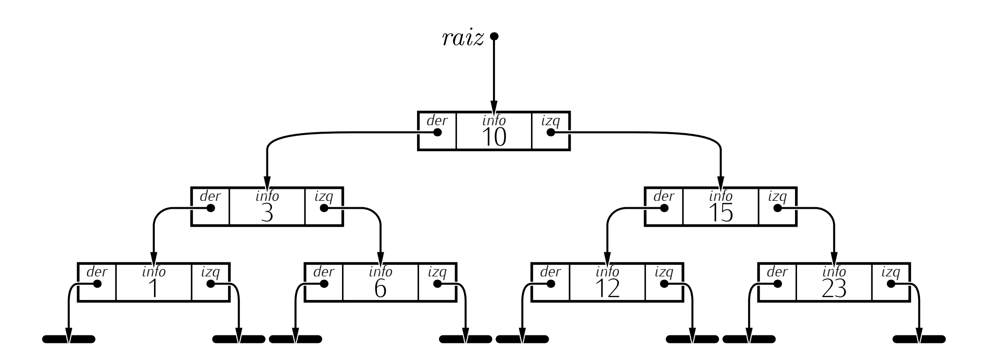
$$\begin{pmatrix} 0 & 0 & 2.5 & 0 & 0 & 1.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.3 & 8.1 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

  De los 100 componentes de esta matriz de 10×10, tan sólo hay 6 no nulos. Las matrices dispersas pueden representarse con listas de listas para ahorrar memoria. Una lista mantiene las filas que, a su vez, son listas de valores no nulos. En estas últimas listas, cada nodo almacena la columna del valor no nulo y el propio valor. La matriz dispersa del ejemplo se representaría así (suponiendo que filas y columnas empiezan numerándose en 1, como es habitual en matemáticas):

El ahorro de memoria es notabilísimo: si un **float** ocupa 8 bytes, hemos pasado de 800 a 132 bytes consumidos. El ahorro es relativamente mayor cuanto mayor es la matriz. Eso sí, la complejidad de los algoritmos que manipulan esa estructura es también notabilísima. ¡Imagina el procedimiento que permite multiplicar eficientemente dos matrices dispersas representadas así!

- Un *árbol binario de búsqueda* es una estructura montada con registros enlazados, pero no es una lista. Cada nodo tiene cero, uno o dos *hijos*: uno a su izquierda y uno a su derecha. Los nodos que no tienen hijos se llaman *hojas*. El nodo más alto, del que descienden todos los demás, se llama nodo *raíz*. Los descendientes de un nodo (sus hijos, nietos, biznietos, etc.) tienen una curiosa propiedad: si descienden por su izquierda, tienen valores más pequeños que el de cualquier ancestro, y si descienden por su derecha, valores mayores. Aquí tienes un ejemplo de árbol binario de búsqueda:



  Una ventaja de los árboles binarios de búsqueda es la rapidez con que pueden resolver la pregunta «¿pertenece un valor determinado al conjunto de valores del árbol?». Hay un método recursivo que recibe un puntero a un nodo y dice:

  - si el puntero vale `NULL`; la respuesta es no;
  - si el valor coincide con el del nodo apuntado, la respuesta es sí;
  - si el valor es menor que el valor del nodo apuntado, entonces la respuesta la conoce el hijo izquierdo, por lo que se le pregunta a él (recursivamente);
  - y si el valor es mayor que el valor del nodo apuntado, entonces la respuesta la conoce el hijo derecho, por lo que se le pregunta a él (recursivamente).

  Ingenioso, ¿no? Observa que muy pocos nodos participan en el cálculo de la respuesta. Si deseas saber, por ejemplo, si el 6 pertenece al árbol de la figura, sólo hay que preguntarle a los nodos que tienen el 10, el 3 y el 6. El resto de nodos no se consultan para nada. Siempre es posible responder a una pregunta de pertenencia en un árbol con n nodos visitando un número de nodos que es, a lo sumo, igual a $1 + \log_2 n$. Rapidísimo. ¿Qué costará, a cambio, insertar o borrar un nodo en el árbol? Cabe pensar que mucho más que un tiempo proporcional al número de nodos, pues la estructura de los enlaces es muy compleja. Pero no es así. Existen procedimientos sofisticados que consiguen efectuar esas operaciones en tiempo proporcional ¡al logaritmo en base 2 del número de nodos!

Hay muchas más estructuras de datos que permiten acelerar sobremanera los programas que gestionan grandes conjuntos de datos. Apenas hemos empezado a conocer y aprendido a manejar las herramientas con las que se construyen los programas: las *estructuras de datos* y los *algoritmos*.

Capítulo 5

Ficheros

—Me temo que sí, señora —dijo Alicia—. No recuerdo las cosas como solía. . .
¡y no conservo el mismo tamaño diez minutos seguidos!

LEWIS CARROLL, *Alicia en el País de las Maravillas.*

Acabamos nuestra introducción al lenguaje C con el mismo objeto de estudio con el que finalizamos la presentación del lenguaje Python: los ficheros. Los ficheros permiten guardar información en un dispositivo de almacenamiento de modo que ésta «sobreviva» a la ejecución de un programa. No te vendría mal repasar los conceptos introductorios a ficheros antes de empezar.

5.1. Ficheros de texto y ficheros binarios

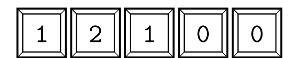
Con Python estudiamos únicamente ficheros de texto. Con C estudiaremos dos tipos de ficheros: ficheros de texto y ficheros binarios.

5.1.1. Representación de la información en los ficheros de texto

Ya conoces los ficheros de texto: contienen datos legibles por una persona y puedes generarlos o modificarlos desde tus propios programas o usando aplicaciones como los editores de texto. Los ficheros binarios, por contra, no están pensados para facilitar su lectura por parte de seres humanos (al menos no directamente).

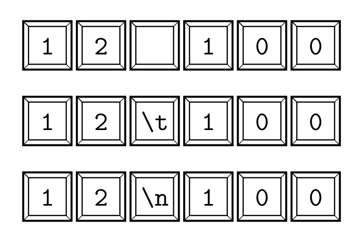
Pongamos que se desea guardar un valor de tipo entero en un fichero de texto, por ejemplo, el valor 12. En el fichero de texto se almacenará el dígito `'1'` (codificado en ASCII como el valor 49) y el dígito `'2'` (codificado en ASCII como el valor 50), es decir, dos datos de tipo **char**. A la hora de leer el dato, podremos leerlo en cualquier variable de tipo entero con capacidad suficiente para almacenar ese valor (un **char**, un **unsigned char**, un **int**, un **unsigned int**, etc.). Esto es así porque la lectura de ese dato pasa por un proceso de interpretación relativamente sofisticado: cuando se lee el carácter `'1'`, se memoriza el valor 1; y cuando se lee el carácter `'2'`, se multiplica por 10 el valor memorizado y se le suma el valor 2. Así se llega al valor 12, que es lo que se almacena en la variable en cuestión. Observa que, codificado como texto, 12 ocupa dos bytes, pero que si se almacena en una variable de tipo **char** ocupa 1 y en una variable de tipo **int** ocupa 4.

Un problema de los ficheros de texto es la necesidad de usar marcas de separación entre sus diferentes elementos. Si, por ejemplo, al valor 12 ha de sucederle el valor 100, no podemos limitarnos a disponer uno a continuación del otro sin más, pues el fichero contendría la siguiente secuencia de caracteres:



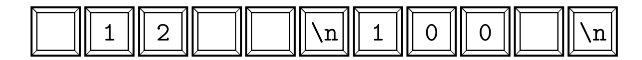
¿Qué estamos representando exactamente? ¿Un 12 seguido de un 100 o un 1 seguido de un 2100? ¿Y por qué no un 1210 seguido de un 0 o, sencillamente, el valor 12100, sin más?

Las marcas de separación son caracteres que decide el programador, pero es corriente que se trate de espacios en blanco, tabuladores o saltos de línea. El valor 12 seguido del valor 100 podría representarse, pues, con cualquiera de estas secuencias de caracteres:



Usar caracteres separadores es fuente, naturalmente, de un coste adicional: un mayor tamaño de los ficheros.

Cuando los separadores son espacios en blanco, es frecuente permitir libertad en cuanto a su número:



Las herramientas con las que leemos los datos de ficheros de texto saben lidiar con las complicaciones que introducen estos separadores blancos repetidos.

Los ficheros de texto cuentan con la ventaja de que se pueden inspeccionar con ayuda de un editor de texto y permiten así, por lo general, deducir el tipo de los diferentes datos que lo componen, pues éstos resultan legibles.

5.1.2. Representación de la información en los ficheros binarios

Los ficheros binarios requieren una mayor precisión en la determinación de la codificación de la información. Si almacenamos el valor 12 en un fichero binario, hemos de decidir si queremos almacenarlo como carácter con o sin signo, como entero con o sin signo, etc. La decisión adoptada determinará la ocupación de la información (uno o cuatro bytes) y su codificación (binario natural o complemento a dos). Si guardamos el 12 como un **char**, guardaremos un solo byte formado por estos 8 bits:

00001100

Pero si optamos por almacenarlo como un **int**, serán cuatro los bytes escritos:

00000000	00000000	00000000	00001100

Un mismo patrón de 8 bits, como

11111111

tiene dos interpretaciones posibles: el valor 255 si entendemos que es un dato de tipo **unsigned char** o el valor −1 si consideramos que codifica un dato de tipo **char**.[1]

[1]Un fichero de texto no presentaría esta ambigüedad: el número se habría escrito como −1 o como 255. Sí que presentaría, sin embargo, un punto de elección reservado al programador: aunque −1 lleva signo y por tanto se almacenará en una variable de algún tipo con signo, ¿queremos almacenarlo en una variable de tipo **char**, una variable de tipo **int** o, por qué no, en una variable de tipo **float**?

Como puedes ver, la secuencia de bits que escribimos en el fichero es exactamente la misma que hay almacenada en la memoria, usando la mismísima codificación binaria. De ahí el nombre de ficheros *binarios*.

. . . . EJERCICIOS . . . .

▶ **310** ¿Qué ocupa en un fichero de texto cada uno de estos datos?

a) 1
b) 0
c) 12
d) -15
e) 128
f) 32767
g) -32768
h) 2147483647
i) -2147483648

¿Y cuánto ocupa cada uno de ellos si los almacenamos en un fichero binario como valores de tipo **int**?

▶ **311** ¿Cómo se interpreta esta secuencia de bytes en cada uno de los siguientes supuestos?

00000000	00000000	00000000	00001100

a) Como cuatro datos de tipo **char**.

b) Como cuatro datos de tipo **unsigned char**.

c) Como un dato de tipo **int**.

d) Como un dato de tipo **unsigned int**.

. . . .

Escribir dos o más datos de un mismo tipo en un fichero binario no requiere la inserción de marcas separadoras: cada cierto número de bytes empieza un nuevo dato (cada cuatro bytes, por ejemplo, empieza un nuevo **int**), así que es fácil decidir dónde empieza y acaba cada dato.

La lectura de un fichero binario requiere un conocimiento exacto del tipo de datos de cada uno de los valores almacenados en él, pues de lo contrario la secuencia de bits carecerá de un significado definido.

Los ficheros binarios no sólo pueden almacenar escalares. Puedes almacenar también registros y vectores pues, a fin de cuentas, no son más que patrones de bits de tamaño conocido. Lo único que no debe almacenarse en ficheros binarios son los punteros. La razón es sencilla: si un puntero apunta a una zona de memoria reservada con *malloc*, su valor es la dirección del primer byte de esa zona. Si guardamos ese valor en disco y lo recuperamos más tarde (en una ejecución posterior, por ejemplo), esa zona puede que no haya sido reservada. Acceder a ella provocará, en consecuencia, un error capaz de abortar la ejecución del programa.

Por regla general, los ficheros binarios son más compactos que los ficheros de texto, pues cada valor ocupa lo mismo que ocuparía en memoria. La lectura (y escritura) de los datos de ficheros binarios es también más rápida, ya que nos ahorramos el proceso de conversión del formato de texto al de representación de información en memoria y viceversa. Pero no todo son ventajas.

5.2. Ficheros de texto

5.2.1. Abrir, leer/escribir, cerrar

Los ficheros de texto se manipulan en C siguiendo el mismo «protocolo» que seguíamos en Python:

> ***Portabilidad de ficheros***
>
> Los ficheros binarios presentan algunos problemas de portabilidad, pues no todos los ordenadores almacenan en memoria los valores numéricos de la misma forma: los ficheros binarios escritos en un ordenador «big-endian» no son directamente legibles en un ordenador «little-endian».
>
> Los ficheros de texto son, en principio, más portables, pues la tabla ASCII es un estándar ampliamente aceptado para el intercambio de ficheros de texto. No obstante, la tabla ASCII es un código de 7 bits que sólo da cobertura a los símbolos propios de la escritura del inglés y algunos caracteres especiales. Los caracteres acentuados, por ejemplo, están excluidos. En los últimos años se ha intentado implantar una familia de estándares que den cobertura a estos y otros caracteres. Como 8 bits resultan insuficientes para codificar todos los caracteres usados en la escritura de cualquier lenguaje, hay diferentes subconjuntos para cada una de las diferentes comunidades culturales. Las lenguas románicas occidentales usan el estándar IsoLatin-1 (o ISO-8859-1), recientemente ampliado con el símbolo del euro para dar lugar al IsoLatin-15 (o ISO-8859-15). Los problemas de portabilidad surgen cuando interpretamos un fichero de texto codificado con IsoLatin-1 como si estuviera codificado con otro estándar: no veremos más que un galimatías de símbolos extraños allí donde se usan caracteres no ASCII.

1. Se abre el fichero en modo lectura, escritura, adición, o cualquier otro modo válido.
2. Se trabaja con él leyendo o escribiendo datos, según el modo de apertura escogido. Al abrir un fichero se dispone un «cabezal» de lectura o escritura en un punto definido del fichero (el principio o el final). Cada acción de lectura o escritura desplaza el cabezal de izquierda a derecha, es decir, de principio a final del fichero.
3. Se cierra el fichero.

Bueno, lo cierto es que, como siempre en C, hay un paso adicional y previo a estos tres: la declaración de una variable de «tipo fichero». La cabecera `stdio.h` incluye la definición de un tipo de datos llamado *FILE* y declara los prototipos de las funciones de manipulación de ficheros. Nuestra variable de tipo fichero ha de ser *un puntero* a *FILE*, es decir, ha de ser de tipo *FILE* *.

Las funciones básicas con las que vamos a trabajar son:

- *fopen*: abre un fichero. Recibe la ruta de un fichero (una cadena) y el modo de apertura (otra cadena) y devuelve un objeto de tipo *FILE* *.

  ```
  FILE * fopen (char ruta[], char modo[]);
  ```

  Los modos de apertura para ficheros de texto con los que trabajaremos son éstos:

  - `"r"` (lectura): El primer carácter leído es el primero del fichero.
  - `"w"` (escritura): Trunca el fichero a longitud 0. Si el fichero no existe, se crea.
  - `"a"` (adición): Es un modo de escritura que preserva el contenido original del fichero. Los caracteres escritos se añaden al final del fichero.

  Si el fichero no puede abrirse por cualquier razón, *fopen* devuelve el valor `NULL`. (Observa que los modos se indican con cadenas, no con caracteres: debes usar comillas dobles.)

- *fclose*: cierra un fichero. Recibe el *FILE* * devuelto por una llamada previa a *fopen*.

  ```
  int fclose (FILE * fichero);
  ```

Modos de apertura para lectura y escritura simultánea

Los modos "r", "w" y "a" no son los únicos válidos para los ficheros de texto. Puedes usar, además, éstos otros: "r+", "w+" y "a+". Todos ellos abren los ficheros en modo de lectura y escritura a la vez. Hay, no obstante, matices que los diferencian:

- "r+": No se borra el contenido del fichero, que debe existir previamente. El «cabezal» de lectura/escritura se sitúa al principio del fichero.
- "w+": Si el fichero no existe, se crea, y si existe, se trunca el contenido a longitud cero. El «cabezal» de lectura/escritura se sitúa al principio del fichero.
- "a+": Si el fichero no existe, se crea. El «cabezal» de lectura/escritura se sitúa al final del fichero.

Una cosa es que existan estos métodos y otra que te recomendemos su uso. Te lo desaconsejamos. Resulta muy difícil escribir en medio de un fichero de texto a voluntad sin destruir la información previamente existente en él, pues cada línea puede ocupar un número de caracteres diferente.

El valor devuelto por *fclose* es un código de error que nos advierte de si hubo un fallo al cerrar el fichero. El valor 0 indica éxito y el valor EOF (predefinido en stdio.h) indica error. Más adelante indicaremos cómo obtener información adicional acerca del error detectado.

Cada apertura de un fichero con *fopen* debe ir acompañada de una llamada a *fclose* una vez se ha terminado de trabajar con el fichero.

- *fscanf*: lee de un fichero. Recibe un fichero abierto con *fopen* (un *FILE* *), una cadena de formato (usando las marcas de formato que ya conoces por *scanf*) y las direcciones de memoria en las que debe depositar los valores leídos. La función devuelve el número de elementos efectivamente leídos (valor que puedes usar para comprobar si la lectura se completó con éxito).

```
int fscanf (FILE * fichero, char formato[], direcciones);
```

- *fprintf*: escribe en un fichero. Recibe un fichero abierto con *fopen* (un *FILE* *), una cadena de formato (donde puedes usar las marcas de formato que aprendiste a usar con *printf*) y los valores que deseamos escribir. La función devuelve el número de caracteres efectivamente escritos (valor que puedes usar para comprobar si se escribieron correctamente los datos).

```
int fprintf (FILE * fichero, char formato[], valores);
```

- *feof*: devuelve 1 si estamos al final del fichero y 0 en caso contrario. El nombre de la función es abreviatura de «end of file» (en español, «fin de fichero»). ¡Ojo! Sólo tiene sentido consultar si se está o no al final de fichero tras efectuar una lectura de datos. (Este detalle complicará un poco las cosas.)

```
int feof (FILE * fichero);
```

Como puedes ver no va a resultar muy difícil trabajar con ficheros de texto en C. A fin de cuentas, las funciones de escritura y lectura son básicamente idénticas a *printf* y *scanf*, y ya hemos aprendido a usarlas. La única novedad destacable es la nueva forma de detectar si hemos llegado al final de un fichero o no: ya no se devuelve la cadena

vacía como consecuencia de una lectura al final del fichero, como ocurría en Python, sino que hemos de preguntar explícitamente por esa circunstancia usando una función (*feof*).

Nada mejor que un ejemplo para aprender a utilizar ficheros de texto en C. Vamos a generar los 1000 primeros números primos y a guardarlos en un fichero de texto. Cada número se escribirá en una línea.

genera_primos.c

genera_primos.c

```
#include <stdio.h>

int es_primo(int n)
{
  int i, j, primo;
  primo = 1;
  for (j=2; j<=n/2; j++)
    if (n % j == 0) {
      primo = 0;
      break;
    }
  return primo;
}

int main(void)
{
  FILE * fp;
  int i, n;

  fp = fopen("primos.txt", "w");
  i = 1;
  n = 0;
  while (n<1000) {
    if (es_primo(i)) {
      fprintf(fp, "%d\n", i);
      n++;
    }
    i++;
  }
  fclose(fp);

  return 0;
}
```

Hemos llamado a la variable de fichero *fp* por ser abreviatura del término «file pointer» (puntero a fichero). Es frecuente utilizar ese nombre para las variables de tipo *FILE* *.

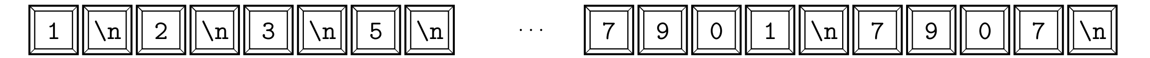
Una vez compilado y ejecutado el programa `genera_primos` obtenemos un fichero de texto llamado `primos.txt` del que te mostramos sus primeras y últimas líneas (puedes comprobar la corrección del programa abriendo el fichero `primos.txt` con un editor de texto):

primos.txt

```
1
2
3
5
7
11
13
17
19
```

```
23
⋮
7823
7829
7841
7853
7867
7873
7877
7879
7883
7901
7907
```

Aunque en pantalla lo vemos como una secuencia de líneas, no es más que una secuencia de caracteres:



Diseñemos ahora un programa que lea el fichero `primos.txt` generado por el programa anterior y muestre por pantalla su contenido:

```
1  #include <stdio.h>
2  
3  int main(void)
4  {
5     FILE * fp;
6     int i;
7  
8     fp = fopen("primos.txt", "r");
9     fscanf(fp, "%d", &i);
10    while ( ! feof(fp) ) {
11       printf("%d\n", i);
12       fscanf(fp, "%d", &i);
13    }
14    fclose(fp);
15 
16    return 0;
17 }
```

Observa que la llamada a *fscanf* se encuentra en un bucle que se lee así «mientras no se haya acabado el fichero…», pues *feof* averigua si hemos llegado al final del fichero. La línea 9 contiene una lectura de datos para que la consulta a *feof* tenga sentido: *feof* sólo actualiza su valor tras efectuar una operación de lectura del fichero. Si no te gusta la aparición de dos sentencias *fscanf*, puedes optar por esta alternativa:

```
#include <stdio.h>

int main(void)
{
   FILE * fp;
   int i;

   fp = fopen("primos.txt", "r");
   while (1) {
      fscanf(fp, "%d", &i);
```

```
      if (feof(fp)) break;
      printf("%d\n", i);
   }
   fclose(fp);

   return 0;
}
```

Y si deseas evitar el uso de **break**, considera esta otra:

lee_primos.c

```c
#include <stdio.h>

int main(void)
{
   FILE * fp;
   int i;

   fp = fopen("primos.txt", "r");
   do {
      fscanf(fp, "%d", &i);
      if (!feof(fp))
         printf("%d\n", i);
   } while (!feof(fp));
   fclose(fp);

   return 0;
}
```

¿Y si el fichero no existe?

Al abrir un fichero puede que detectes un error: *fopen* devuelve la dirección `NULL`. Hay varias razones, pero una que te ocurrirá al probar algunos de los programas del texto es que el fichero que se pretende leer no existe. Una solución puede consistir en crearlo en ese mismo instante:

```c
f = fopen(ruta, "r");
if (f == NULL) {
   f = fopen(ruta, "w");
   fclose(f);
   f = fopen(ruta, "r");
}
```

Si el problema era la inexistencia del fichero, este truco funcionará, pues el modo `"w"` lo crea cuando no existe.

Es posible, no obstante, que incluso este método falle. En tal caso, es probable que tengas un problema de permisos: ¿tienes permiso para leer ese fichero?, ¿tienes permiso para escribir en el directorio en el que reside o debe residir el fichero? Más adelante prestaremos atención a esta cuestión.

. . . . EJERCICIOS . . . .

▶ **312** Diseña un programa que añada al fichero `primos.txt` los 100 siguientes números primos. El programa leerá el contenido actual del fichero para averiguar cuál es el último primo del fichero. A continuación, abrirá el fichero en modo adición (`"a"`) y añadirá 100 nuevos primos. Si ejecutásemos una vez `genera_primos` y, a continuación, dos veces el nuevo programa, el fichero acabaría conteniendo los 1200 primeros primos.

▶ 313 Diseña un programa que lea de teclado una frase y escriba un fichero de texto llamado `palabras.txt` en el que cada palabra de la frase ocupa una línea.

▶ 314 Diseña un programa que lea de teclado una frase y escriba un fichero de texto llamado `letras.txt` en el que cada línea contenga un carácter de la frase.

▶ 315 Modifica el programa miniGalaxis para que gestione una lista de records. Un fichero de texto, llamado `minigalaxis.records` almacenará el nombre y número de movimientos de los 5 mejores jugadores de todos los tiempos (los que completaron el juego usando el menor número de sondas).

▶ 316 Disponemos de dos ficheros: uno contiene un diccionario y el otro, un texto. El diccionario está ordenado alfabéticamente y contiene una palabra en cada línea. Diseña un programa que lea el diccionario en un vector de cadenas y lo utilice para detectar errores en el texto. El programa mostrará por pantalla las palabras del texto que no están en el diccionario, indicando los números de línea en que aparecen.

Supondremos que el diccionario contiene, a lo sumo, 1000 palabras y que la palabra más larga (tanto en el diccionario como en el texto) ocupa 30 caracteres.

(Si quieres usar un diccionario real como el descrito y trabajas en Unix, encontrarás uno en inglés en `/usr/share/dict/words` o `/usr/dict/words`. Puedes averiguar el número de palabras que contiene con el comando `wc` de Unix.)

▶ 317 Modifica el programa del ejercicio anterior para que el número de palabras del vector que las almacena se ajuste automáticamente al tamaño del diccionario. Tendrás que usar memoria dinámica.

Si usas un vector de palabras, puedes efectuar dos pasadas de lectura en el fichero que contiene el diccionario: una para contar el número de palabras y saber así cuánta memoria es necesaria y otra para cargar la lista de palabras en un vector dinámico. Naturalmente, antes de la segunda lectura deberás haber reservado la memoria necesaria.

Una alternativa a leer dos veces el fichero consiste en usar *realloc* juiciosamente: reserva inicialmente espacio para, digamos, 1000 palabras; si el diccionario contiene un número de palabras mayor que el que cabe en el espacio de memoria reservada, duplica la capacidad del vector de palabras (cuantas veces sea preciso si el problema se da más de una vez).

Otra posibilidad es usar una lista simplemente enlazada, pues puedes crearla con una primera lectura. Sin embargo, no es recomendable que sigas esta estrategia, pues no podrás efectuar una búsqueda dicotómica a la hora de determinar si una palabra está incluida o no en el diccionario.

. . . .

Ya vimos en su momento que *fscanf* presenta un problema cuando leemos cadenas: sólo lee una «palabra», es decir, se detiene al llegar a un blanco. Aprendimos a usar entonces una función, *gets*, que leía una línea completa. Hay una función equivalente para ficheros de texto:

```
char * fgets(char cadena[], int max_tam, FILE * fichero);
```

¡Ojo con el prototipo de *fgets*! ¡El parámetro de tipo *FILE* * es el último, no el primero! Otra incoherencia de C. El primer parámetro es la cadena en la que se desea depositar el resultado de la lectura. El segundo parámetro, un entero, es una medida de seguridad: es el máximo número de bytes que queremos leer en la cadena. Ese límite permite evitar peligrosos desbordamientos de la zona de memoria reservada para *cadena* cuando la cadena leída es más larga de lo previsto. El último parámetro es, finalmente, el fichero del que vamos a leer (previamente se ha abierto con *fopen*). La función se ocupa de terminar correctamente la cadena leída con un `'\0'`, *pero respetando el salto de línea* (`\n`) *si lo hubiera.*[2] En caso de querer suprimir el retorno de línea, puedes invocar una función como ésta sobre la cadena leída:

[2]En esto se diferencia de *gets*.

```c
void quita_fin_de_linea(char linea[])
{
  int i;
  for (i=0; linea[i] != '\0'; i++)
    if (linea[i] == '\n') {
      linea[i] = '\0';
      break;
    }
}
```

La función *fgets* devuelve una cadena (un **char** *). En realidad, es un puntero a la propia variable *cadena* cuando todo va bien, y `NULL` cuando no se ha podido efectuar la lectura. El valor de retorno es útil, únicamente, para hacer detectar posibles errores tras llamar a la función.

Hay más funciones de la familia *get*. La función *fgetc*, por ejemplo, lee un carácter:

```c
  int fgetc(FILE * fichero);
```

No te equivoques: devuelve un valor de tipo **int**, pero es el valor ASCII de un carácter. Puedes asignar ese valor a un **unsigned char**, excepto cuando vale `EOF` (de «end of file»), que es una constante (cuyo valor es −1) que indica que no se pudo leer el carácter requerido porque llegamos al final del fichero.

Las funciones *fgets* y *fgetc* se complementan con *fputs* y *fputc*, que en lugar de leer una cadena o un carácter, *escriben* una cadena o un carácter en un fichero abierto para escritura o adición. He aquí sus prototipos:

```c
  int fputs(char cadena[], FILE * fichero);
  int fputc(int caracter, FILE * fichero);
```

Al escribir una cadena con *fputs*, el terminador `'\0'` no se escribe en el fichero. Pero no te preocupes: *fgets* «lo sabe» y lo introduce automáticamente en el vector de caracteres al leer del fichero.

. . . . EJERCICIOS . . . .

▶ 318 Hemos escrito este programa para probar nuestra comprensión de *fgets* y *fputs* (presta atención también a los blancos, que se muestran con el carácter ␣):

```c
#include <stdio.h>
#include <string.h>

#define MAXLON 100

int main (void)
{
  FILE * f;
  char s[MAXLON+1];
  char * aux;

  f = fopen("prueba.txt", "w");
  fputs("si", f);
  fputs("no\n", f);
  fclose(f);

  f = fopen("prueba.txt", "r");
  aux = fgets(s, MAXLON, f);
  printf("%s␣%s\n", aux, s);
  aux = fgets(s, MAXLON, f);
  printf("%s␣%s\n", aux, s);
  fclose(f);

```

```
    return 0;
}
```

Primera cuestión: ¿Cuántos bytes ocupa el fichero *prueba.txt*?

Al ejecutarlo, obtenemos este resultado en pantalla:

```
sino
␣sino

(null)␣sino
```

Segunda cuestión: ¿Puedes explicar con detalle qué ha ocurrido? (El texto «(null)» es escrito automáticamente por *printf* cuando se le pasa como cadena un puntero a NULL.)

5.2.2. Aplicaciones: una agenda y un gestor de una colección de discos compactos

Lo aprendido nos permite ya diseñar programas capaces de escribir y leer colecciones de datos en ficheros de texto.

Una agenda

Vamos a desarrollar un pequeño ejemplo centrado en las rutinas de entrada/salida para la gestión de una agenda montada con una lista simplemente enlazada. En la agenda, que cargaremos de un fichero de texto, tenemos el nombre, la dirección y el teléfono de varias personas. Cada entrada en la agenda se representará con tres líneas del fichero de texto. He aquí un ejemplo de fichero con este formato:

agenda.txt

```
Juan␣Gil
Ronda␣Mijares,␣1220
964␣123456
Ana␣García
Plaza␣del␣Sol,␣13
964-872777
Pepe␣Pérez
Calle␣de␣Arriba,␣1
964␣263␣263
```

Nuestro programa podrá leer en memoria los datos de un fichero como éste y también escribirlos en fichero desde memoria.

Las estructuras de datos que manejaremos en memoria se definen así:

```
struct Entrada {
    char * nombre;
    char * direccion;
    char * telefono;
};

struct NodoAgenda {
    struct Entrada datos;
    struct NodoAgenda * sig;
};

typedef struct NodoAgenda * TipoAgenda;
```

Al final del apartado presentamos el programa completo. Centrémonos ahora en las funciones de escritura y lectura del fichero. La rutina de escritura de datos en un fichero recibirá la estructura y el nombre del fichero en el que guardamos la información. Guardaremos cada entrada de la agenda en tres líneas: una por cada campo.

```
void escribe_agenda(TipoAgenda agenda, char nombre_fichero[])
{
  struct NodoAgenda * aux;
  FILE * fp;

  fp = fopen(nombre_fichero, "w");
  for (aux=agenda; aux!=NULL; aux=aux->sig)
    fprintf(fp, "%s\n%s\n%s\n", aux->datos.nombre,
                                aux->datos.direccion,
                                aux->datos.telefono);
  fclose(fp);
}
```

La lectura del fichero será sencilla:

```
TipoAgenda lee_agenda(char nombre_fichero[])
{
  TipoAgenda agenda;
  struct Entrada * entrada_leida;
  FILE * fp;
  char nombre[MAXCADENA+1], direccion[MAXCADENA+1], telefono[MAXCADENA+1];
  int longitud;

  agenda = crea_agenda();

  fp = fopen(nombre_fichero, "r");
  while (1) {
    fgets(nombre, MAXCADENA, fp);
    if (feof(fp)) break; // Si se acabó el fichero, acabar la lectura.
    quita_fin_de_linea(nombre);

    fgets(direccion, MAXCADENA, fp);
    quita_fin_de_linea(direccion);

    fgets(telefono, MAXCADENA, fp);
    quita_fin_de_linea(telefono);

    agenda = anyadir_entrada(agenda, nombre, direccion, telefono);
  }
  fclose(fp);

  return agenda;
}
```

La única cuestión reseñable es la purga de saltos de línea innecesarios.

He aquí el listado completo del programa:

agenda_sencilla.c   agenda_sencilla.c

```
#include <stdio.h>
#include <stdlib.h>

#define MAXCADENA 200

enum { Ver=1, Alta, Buscar, Salir };

```

```
struct Entrada {
   char * nombre;
   char * direccion;
   char * telefono;
};

struct NodoAgenda {
   struct Entrada datos;
   struct NodoAgenda * sig;
};

typedef struct NodoAgenda * TipoAgenda;

void quita_fin_de_linea(char linea[])
{
   int i;
   for (i=0; linea[i] != '\0'; i++)
      if (linea[i] == '\n') {
         linea[i] = '\0';
         break;
      }
}

void muestra_entrada(struct NodoAgenda * e)
   // Podríamos haber pasado e por valor, pero resulta más eficiente (y no mucho más
   // incómodo) hacerlo por referencia: pasamos así sólo 4 bytes en lugar de 12.
{
   printf("Nombre␣␣␣:␣%s\n", e->datos.nombre);
   printf("Dirección:␣%s\n", e->datos.direccion);
   printf("Teléfono␣:␣%s\n", e->datos.telefono);
}

void libera_entrada(struct NodoAgenda * e)
{
   int i;

   free(e->datos.nombre);
   free(e->datos.direccion);
   free(e->datos.telefono);
   free(e);
}


TipoAgenda crea_agenda(void)
{
   return NULL;
}

TipoAgenda anyadir_entrada(TipoAgenda agenda, char nombre[],
                               char direccion[], char telefono[])
{
   struct NodoAgenda * aux, * e;

   /* Averiguar si ya tenemos una persona con ese nombre */
   if (buscar_entrada_por_nombre(agenda, nombre) != NULL)
      return agenda;

   /* Si llegamos aquí, es porque no teníamos registrada a esa persona. */
   e = malloc(sizeof(struct NodoAgenda));
```

```
  e->datos.nombre = malloc((strlen(nombre)+1)*sizeof(char));
  strcpy(e->datos.nombre, nombre);
  e->datos.direccion = malloc((strlen(direccion)+1)*sizeof(char));
  strcpy(e->datos.direccion, direccion);
  e->datos.telefono = malloc((strlen(telefono)+1)*sizeof(char));
  strcpy(e->datos.telefono, telefono);
  e->sig = agenda;
  agenda = e;
  return agenda;
}

void muestra_agenda(TipoAgenda agenda)
{
  struct NodoAgenda * aux;

  for (aux = agenda; aux != NULL; aux = aux->sig)
    muestra_entrada(aux);
}

struct NodoAgenda * buscar_entrada_por_nombre(TipoAgenda agenda, char nombre[])
{
  struct NodoAgenda * aux;

  for (aux = agenda; aux != NULL; aux = aux->sig)
    if (strcmp(aux->datos.nombre, nombre) == 0)
      return aux;

  return NULL;
}

void libera_agenda(TipoAgenda agenda)
{
  struct NodoAgenda * aux, *siguiente;

  aux = agenda;
  while (aux != NULL) {
    siguiente = aux->sig;
    libera_entrada(aux);
    aux = siguiente;
  }
}

void escribe_agenda(TipoAgenda agenda, char nombre_fichero[])
{
  struct NodoAgenda * aux;
  FILE * fp;

  fp = fopen(nombre_fichero, "w");
  for (aux=agenda; aux!=NULL; aux=aux->sig)
    fprintf(fp, "%s\n%s\n%s\n", aux->datos.nombre,
                                aux->datos.direccion,
                                aux->datos.telefono);
  fclose(fp);
}

TipoAgenda lee_agenda(char nombre_fichero[])
{
  TipoAgenda agenda;
  struct Entrada * entrada_leida;
```

```
  FILE * fp;
  char nombre[MAXCADENA+1], direccion[MAXCADENA+1], telefono[MAXCADENA+1];
  int longitud;

  agenda = crea_agenda();

  fp = fopen(nombre_fichero, "r");
  while (1) {
    fgets(nombre, MAXCADENA, fp);
    if (feof(fp)) break; // Si se acabó el fichero, acabar la lectura.
    quita_fin_de_linea(nombre);

    fgets(direccion, MAXCADENA, fp);
    quita_fin_de_linea(direccion);

    fgets(telefono, MAXCADENA, fp);
    quita_fin_de_linea(telefono);

    agenda = anyadir_entrada(agenda, nombre, direccion, telefono);
  }
  fclose(fp);

  return agenda;
}


/************************************************************************
 * Programa principal
 ************************************************************************/

int main(void)
{
  TipoAgenda miagenda;
  struct NodoAgenda * encontrada;
  int opcion;
  char nombre[MAXCADENA+1];
  char direccion[MAXCADENA+1];
  char telefono[MAXCADENA+1];
  char linea[MAXCADENA+1];

  miagenda = lee_agenda("agenda.txt");

  do {
    printf("Menú:\n");
    printf("1) Ver contenido completo de la agenda.\n");
    printf("2) Dar de alta una persona.\n");
    printf("3) Buscar teléfonos de una persona.\n");
    printf("4) Salir.\n");
    printf("Opción: ");
    gets(linea); sscanf(linea, "%d", &opcion);

    switch(opcion) {

      case Ver:
        muestra_agenda(miagenda);
        break;

      case Alta:
        printf("Nombre   : "); gets(nombre);
```

```
            printf ("Dirección:␣"); gets(direccion);
            printf ("Teléfono␣:␣"); gets(telefono);
            miagenda = anyadir_entrada(miagenda, nombre, direccion, telefono);
            break;

        case Buscar:
            printf ("Nombre:␣"); gets(nombre);
            encontrada = buscar_entrada_por_nombre(miagenda, nombre);
            if (encontrada == NULL)
              printf ("No␣hay␣nadie␣llamado␣%s␣en␣la␣agenda.\n", nombre);
            else
              muestra_entrada(encontrada);
            break;
      }
    } while (opcion != Salir);


    escribe_agenda(miagenda, "agenda.txt");
    libera_agenda(miagenda);

    return 0;
}
```

Entrada/salida de fichero para el programa de gestión de una colección de discos

Acabamos esta sección dedicada a los ficheros de texto con una aplicación práctica. Vamos a añadir funcionalidad al programa desarrollado en el apartado 4.11: el programa cargará la «base de datos» tan pronto inicie su ejecución leyendo un fichero de texto y la guardará en el mismo fichero, recogiendo los cambios efectuados, al final.

En primer lugar, discutamos brevemente acerca del formato del fichero de texto. Podemos almacenar cada dato en una línea, así:

discoteca.txt
```
Expression
John␣Coltrane
1972
Ogunde
To␣be
Offering
Expression
Number␣One
Logos
Tangerine␣Dream
1982
Logos
Dominion
Ignacio
Vangelis
1977
Ignacio
```

Pero hay un serio problema: ¿cómo sabe el programa dónde empieza y acaba cada disco? El programa no puede distinguir entre el título de una canción, el de un disco o el nombre de un intérprete. Podríamos marcar cada línea con un par de caracteres que nos indiquen qué tipo de información mantiene:

discoteca.txt
```
TD␣Expression
```

```
IN␣John␣Coltrane
AÑ␣1972
TC␣Ogunde
TC␣To␣be
TC␣Offering
TC␣Expression
TC␣Number␣One
TD␣Logos
IN␣Tangerine␣Dream
AÑ␣1982
TC␣Logos
TC␣Dominion
TD␣Ignacio
IN␣Vangelis
AÑ␣1977
TC␣Ignacio
```

Con `TD` indicamos «título de disco»; con `IN`, «intérprete»; con `AÑ`, «año»; y con `TC`, «título de canción». Pero esta solución complica las cosas en el programa: no sabemos de qué tipo es una línea hasta haber leído sus dos primeros caracteres. O sea, sabemos que un disco «ha acabado» cuando ya hemos leído una línea del siguiente. No es que no se pueda trabajar así, pero resulta complicado. Como podemos definir libremente el formato, optaremos por uno que preceda los títulos de las canciones por un número que indique cuántas canciones hay:

discoteca.txt   discoteca.txt

```
Expression
John␣Coltrane
1972
5
Ogunde
To␣be
Offering
Expression
Number␣One
Logos
Tangerine␣Dream
1982
2
Logos
Dominion
Ignacio
Vangelis
1977
1
Ignacio
```

La lectura de la base de datos es relativamente sencilla:

```
void quita_fin_de_linea(char linea[])
{
  int i;
  for (i=0; linea[i] != '\0'; i++)
    if (linea[i] == '\n') {
      linea[i] = '\0';
      break;
    }
}
```

```
TipoColeccion carga_coleccion(char nombre_fichero[])
{
   FILE * f;
   char titulo_disco[MAXCAD+1], titulo_cancion[MAXCAD+1], interprete[MAXCAD+1];
   char linea[MAXCAD+1];
   int anyo;
   int numcanciones;
   int i;
   TipoColeccion coleccion;
   TipoListaCanciones lista_canciones;

   coleccion = crea_coleccion();
   f = fopen(nombre_fichero, "r");
   while(1) {
      fgets(titulo_disco, MAXCAD, f);
      if (feof(f))
         break;
      quita_fin_de_linea(titulo_disco);
      fgets(interprete, MAXCAD, f);
      quita_fin_de_linea(interprete);
      fgets(linea, MAXCAD, f); sscanf(linea, "%d", &anyo);
      fgets(linea, MAXCAD, f); sscanf(linea, "%d", &numcanciones);
      lista_canciones = crea_lista_canciones();
      for (i=0; i<numcanciones; i++) {
         fgets(titulo_cancion, MAXCAD, f);
         quita_fin_de_linea(titulo_cancion);
         lista_canciones = anyade_cancion(lista_canciones, titulo_cancion);
      }
      coleccion = anyade_disco(coleccion, titulo_disco, interprete, anyo, lista_canciones);
   }
   fclose(f);

   return coleccion;
}
```

Tan sólo cabe reseñar dos cuestiones:

- La detección del final de fichero se ha de hacer tras una lectura infructuosa, por lo que la hemos dispuesto tras el primer *fgets* del bucle.
- La lectura de líneas con *fgets* hace que el salto de línea esté presente, así que hay que eliminarlo explícitamente.

Al guardar el fichero hemos de asegurarnos de que escribimos la información en el mismo formato:

```
void guarda_coleccion(TipoColeccion coleccion, char nombre_fichero[])
{
   struct Disco * disco;
   struct Cancion * cancion;
   int numcanciones;
   FILE * f;

   f = fopen(nombre_fichero, "w");
   for (disco = coleccion; disco != NULL; disco = disco->sig) {
      fprintf(f, "%s\n", disco->titulo);
      fprintf(f, "%s\n", disco->interprete);
      fprintf(f, "%d\n", disco->anyo);

```

```
    numcanciones = 0;
    for (cancion = disco->canciones; cancion != NULL; cancion = cancion->sig)
      numcanciones++;
    fprintf(f, "%d\n", numcanciones);

    for (cancion = disco->canciones; cancion != NULL; cancion = cancion->sig)
      fprintf(f, "%s\n", cancion->titulo);
  }
  fclose(f);
}
```

Observa que hemos recorrido dos veces la lista de canciones de cada disco: una para saber cuántas canciones contiene (y así poder escribir en el fichero esa cantidad) y otra para escribir los títulos de las canciones.

Aquí tienes las modificaciones hechas al programa principal:

discoteca2_1.c **discoteca2.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
⋮


int main(void)
{
  int opcion;
  TipoColeccion coleccion;
  char titulo_disco[MAXCAD+1], titulo_cancion[MAXCAD+1], interprete[MAXCAD+1];
  char linea[MAXCAD+1];
  int anyo;
  struct Disco * undisco;
  TipoListaCanciones lista_canciones;

  coleccion = carga_coleccion("discoteca.txt");

  do {
    printf("Menú\n");
    printf("1)␣Añadir␣disco\n");
    printf("2)␣Buscar␣por␣título␣del␣disco\n");
    printf("3)␣Buscar␣por␣intérprete\n");
    printf("4)␣Buscar␣por␣título␣de␣canción\n");
    printf("5)␣Mostrar␣todo\n");
    printf("6)␣Eliminar␣un␣disco␣por␣título␣e␣intérprete\n");
    printf("7)␣Finalizar\n");
    printf("Opción:␣"); gets(linea); sscanf(linea, "%d", &opcion);
⋮

  guarda_coleccion(coleccion, "discoteca.txt");
  coleccion = libera_coleccion(coleccion);

  return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 319 La gestión de ficheros mediante su carga previa en memoria puede resultar problemática al trabajar con grandes volúmenes de información. Modifica el programa de

la agenda para que no cargue los datos en memoria. Todas las operaciones (añadir datos y consultar) se efectuarán gestionando directamente ficheros.

▶ 320 Modifica el programa propuesto en el ejercicio anterior para que sea posible borrar entradas de la agenda. (Una posible solución pasa por trabajar con dos ficheros, uno original y uno para copias, de modo que borrar una información sea equivalente a no escribirla en la copia.)

▶ 321 Modifica el programa de la agenda para que se pueda mantener más de un teléfono asociado a una persona. El formato del fichero pasa a ser el siguiente:

- Una línea que empieza por la letra N contiene el nombre de una persona.
- Una línea que empieza por la letra D contiene la dirección de la persona cuyo nombre acaba de aparecer.
- Una línea que empieza por la letra T contiene un número de teléfono asociado a la persona cuyo nombre apareció más recientemente en el fichero.

Ten en cuenta que no se puede asociar más de una dirección a una persona (y si eso ocurre en el fichero, debes notificar la existencia de un error), pero sí más de un teléfono. Además, puede haber líneas en blanco (o formadas únicamente por espacios en blanco) en el fichero. He aquí un ejemplo de fichero con el nuevo formato:

agenda.txt

```
N␣Juan␣Gil
D␣Ronda␣Mijares,␣1220
T␣964␣123456

N␣Ana␣García
D␣Plaza␣del␣Sol,␣13
T␣964-872777
T␣964-872778


N␣Pepe␣Pérez
D␣Calle␣de␣Arriba,␣1
T␣964␣263␣263
T␣964␣163␣163
T␣96␣2663␣663
```

▶ 322 En un fichero *matriz.mat* almacenamos los datos de una matriz de enteros con el siguiente formato:

- La primera línea contiene el número de filas y columnas.
- Cada una de las restantes líneas contiene tantos enteros (separados por espacios) como indica el número de columnas. Hay tantas líneas de este estilo como filas tiene la matriz.

Este ejemplo define una matriz de 3×4 con el formato indicado:

matriz.txt

```
3␣4
1␣␣0␣␣3␣␣4
0␣-1␣12␣-1
3␣␣0␣99␣-3
```

Escribe un programa que lea *matriz.mat* efectuando las reservas de memoria dinámica que corresponda y muestre por pantalla, una vez cerrado el fichero, el contenido de la matriz.

▶ 323 Modifica el programa del ejercicio anterior para que, si hay menos líneas con valores de filas que filas declaradas en la primera línea, se rellene el restante número de filas con valores nulos.

Aquí tienes un ejemplo de fichero con menos filas que las declaradas:

matriz_incompleta.txt

```
3␣4
1␣0␣3␣4␣␣␣␣␣␣
```

▶ 324 Diseña un programa que facilite la gestión de una biblioteca. El programa permitirá prestar libros. De cada libro se registrará al menos el título y el autor. En cualquier instante se podrá volcar el estado de la biblioteca a un fichero y cargarlo de él.

Conviene que la biblioteca sea una lista de nodos, cada uno de los cuales representa un libro. Uno de los campos del libro podría ser una cadena con el nombre del prestatario. Si dicho nombre es la cadena vacía, se entenderá que el libro está disponible.

Permisos Unix

Los ficheros Unix llevan asociados unos permisos con los que es posible determinar qué usuarios pueden efectuar qué acciones sobre cada fichero. Las acciones son: leer, escribir y ejecutar (esta última limitada a ficheros ejecutables, es decir, resultantes de una compilación o que contienen código fuente de un lenguaje interpretado y siguen cierto convenio). Se puede fijar cada permiso para el usuario «propietario» del fichero, para los usuarios de su mismo grupo o para todos los usuarios del sistema.

Cuando ejecutamos el comando `ls` con la opción `-l`, podemos ver los permisos codificados con las letras `rwx` y el carácter `-`:

```
-rw-r--r--    1 usuario  migrupo        336 may 12 10:43 kk.c
-rwxr-x---    1 usuario  migrupo      13976 may 12 10:43 a.out
```

El fichero `kk.c` tiene permiso de lectura y escritura para el usuario (caracteres 2 a 4), de sólo lectura para los usuarios de su grupo (caracteres 5 a 7) y de sólo lectura para el resto de usuarios (caracteres 8 a 10). El fichero `a.out` puede ser leído, modificado y ejecutado por el usuario. Los usuarios del mismo grupo pueden leerlo y ejecutarlo, pero no modificar su contenido. El resto de usuarios no puede acceder al fichero.

El comando Unix `chmod` permite modificar los permisos de un fichero. Una forma tradicional de hacerlo es con un número octal que codifica los permisos. Aquí tienes un ejemplo de uso:

```
$ chown 0700 a.out ↲
$ ls -l a.out ↲
-rwx------    1 usuario  migrupo      13976 may 12 10:43 a.out
```

El valor octal 0700 (que en binario es `111000000`), por ejemplo, otorga permisos de lectura, escritura y ejecución al propietario del fichero, y elimina cualquier permiso para el resto de usuarios. De cada 3 bits, el primero fija el permiso de lectura, el segundo el de escritura y el tercero el de ejecución. Los 3 primeros bits corresponden al usuario, los tres siguientes al grupo y los últimos 3 al resto. Así pues, 0700 equivale a `-rwx---` en la notación de `ls -l`.

Por ejemplo, para que `a.out` sea también legible y ejecutable por parte de cualquier miembro del grupo del propietario puedes usar el valor 0750 (que equivale a `-rwxr-x--`).

5.2.3. Los «ficheros» de consola

Hay tres ficheros de texto predefinidos y ya abiertos cuando se inicia un programa: los «ficheros» de consola. En realidad, no son ficheros, sino *dispositivos*:

- *stdin* (entrada estándar): el teclado;
- *stdout* (salida estándar): la pantalla;
- *stderr* (salida estándar *de error*): ¿?

¿Qué es *stderr*? En principio es también la pantalla, pero podría ser, por ejemplo un fichero en el que deseamos llevar un cuaderno de bitácora con las anomalías o errores detectados durante la ejecución del programa.

La función *printf* es una forma abreviada de llamar a *fprintf* sobre *stdout* y *scanf* encubre una llamada a *fscanf* sobre *stdin*. Por ejemplo, estas dos llamadas son equivalentes:

```
printf("Esto es la %s\n", "pantalla");
fprintf(stdout, "Esto es la %s\n", "pantalla");
```

El hecho de que, en el fondo, Unix considere al teclado y la pantalla equivalentes a ficheros nos permite hacer ciertas cosas curiosas. Por ejemplo, si deseamos ejecutar un programa cuyos datos se deben leer de teclado o de fichero, según convenga, podemos decidir la fuente de entrada en el momento de la ejecución del programa. Este programa, por ejemplo, permite elegir al usuario entre leer de teclado o leer de fichero:

selecciona_entrada.c

```c
#include <stdio.h>

int main(void)
{
  FILE * fp;
  char dedonde[80], nombre[80];
  int n;

  printf("Leo de fichero o de teclado (f/t)?: ");
  gets(dedonde);
  if (dedonde[0] == 'f') {
    printf("Nombre del fichero: ");
    gets(nombre);
    fp = fopen(nombre, "r");
  }
  else
    fp = stdin;

  ...
  fscanf(fp, "%d", &n); /* Lee de fichero o teclado. */
  ...
  if (fp != stdin)
    fclose(fp);
  ...

  return 0;
}
```

Existe otra forma de trabajar con fichero o teclado que es más cómoda para el programador: usando la capacidad de *redirección* que facilita el intérprete de comandos Unix. La idea consiste en desarrollar el programa considerando sólo la lectura por teclado y, cuando iniciamos la ejecución del programa, *redirigir* un fichero al teclado. Ahora verás cómo. Fíjate en este programa:

De cadena a entero o flotante

Los ficheros de texto contienen eso, texto. No obstante, el texto se interpreta en ocasiones como si codificara valores enteros o flotantes. La función *fscanf*, por ejemplo, es capaz de leer texto de un fichero e interpretarlo como si fuera un entero o un flotante. Cuando hacemos `fscanf(f, "%d", &a)`, donde *a* es de tipo **int**, se leen caracteres del fichero y se interpretan como un entero. Pero hay un problema potencial: el texto puede no corresponder a un valor entero, con lo que la lectura no se efectuaría correctamente. Una forma de curarse en salud es leer como cadena los siguientes caracteres (con *fscanf* y la marca de formato `%s` o con *gets*, por ejemplo), comprobar que la secuencia de caracteres leída describe un entero (o un flotante, según convenga) y convertir ese texto en un entero (o flotante). ¿Cómo efectuar la conversión? C nos ofrece en su biblioteca estándar la función *atoi*, que recibe una cadena y devuelve un entero. Has de incluir la cabecera `stdlib.h` para usarla. Aquí tienes un ejemplo de uso:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char a[] = "123";
  int b;

  b = atoi(a);

  printf("La cadena %s se interpreta como el entero %d con atoi\n",
         a, b);

  return 0;
}
```

Si deseas interpretar el texto como un **float**, puedes usar *atof* en lugar de *atoi*. Así de fácil.

pares.c

```c
#include <stdio.h>

int main(void)
{
  int i, n;

  for (i=0; i<10; i++) {
    scanf("%d", &n);
    if (n%2==0)
      printf("[%d]", n);
  }

  return 0;
}
```

Si lo compilas para generar un programa *pares*, lo ejecutas e introduces los siguientes 10 números enteros, obtendrás este resultado en pantalla:

```
$ pares ↲
3
5
6
[6]
```

```
7
2
[2]
10
[10]
2
[2]
1
3
13
```

Cada vez que el ordenador ha detectado un número par, lo ha mostrado en pantalla entre corchetes.

Creemos ahora, con la ayuda de un editor de texto, `numeros.txt`, un fichero de texto con los mismos 10 números enteros que hemos introducido por teclado antes:

numeros.txt

```
3
5
6
7
2
10
2
1
3
13
```

Podemos llamar a *pares* así:

```
$ pares < numeros.txt ↲
[6]
[2]
[10]
[2]
```

El carácter `<` indica a Unix que lea del fichero `numeros.txt` en lugar de leer del teclado. El programa, sin tocar una sola línea, pasa a leer los valores de `numeros.txt` y muestra por pantalla los que son pares.

También podemos redirigir la salida (la pantalla) a un fichero. Fíjate:

```
$ pares < numeros.txt > solopares.txt ↲
```

Ahora el programa se ejecuta sin mostrar texto alguno por pantalla y el fichero *solopares.txt* acaba conteniendo lo que debiera haberse mostrado por pantalla.

```
$ cat solopares.txt ↲
[6]
[2]
[10]
[2]
```

Para redirigir la salida de errores, puedes usar el par de caracteres `2>` seguido del nombre del fichero en el que se escribirán los mensajes de error.

La capacidad de redirigir los dispositivos de entrada, salida y errores tiene infinidad de aplicaciones. Una evidente es automatizar la fase de pruebas de un programa durante su desarrollo. En lugar de escribir cada vez todos los datos que solicita un programa para ver si efectúa correctamente los cálculos, puedes preparar un fichero con los datos de entrada y utilizar redirección para que el programa los lea automáticamente.

Los peligros de gets… y cómo superarlos

Ya habrás podido comprobar que *gets* no es una función segura, pues siempre es posible desbordar la memoria reservada leyendo una cadena suficientemente larga. Algunos compiladores generan una advertencia cuando detectan el uso de *gets*. ¿Cómo leer, pues, una línea de forma segura? Una posibilidad consiste en escribir nuestra propia función de lectura carácter a carácter (con ayuda de la función *fgetc*) e imponer una limitación al número de caracteres leídos.

```
int lee_linea(char linea[], int max_lon)
{
  int c, nc = 0;
  max_lon--; /* Se reserva un carácter para el \0 */

  while ( (c = fgetc(stdin)) != EOF ) {
    if (c == '\n')
      break;
    if (nc < max_lon)
      linea[nc++] = c;
  }

  if (c == EOF && nc == 0)
    return EOF;

  linea[nc] = '\0';
  return nc;
}
```

Para leer una cadena en un vector de caracteres con una capacidad máxima de 100 caracteres, haremos:

```
lee_linea(cadena, 100);
```

El valor de *cadena* se modificará para contener la cadena leída. La cadena más larga leída tendrá una longitud de 99 caracteres (recuerda que el `'\0'` ocupa uno de los 100).

Pero hay una posibilidad aún más sencilla: usar *fgets* sobre *stdin*:

```
fgets(cadena, 100, stdin);
```

Una salvedad: *fgets* incorpora a la cadena leída el salto de línea, cosa que *gets* no hace.

La primera versión, no obstante, sigue teniendo interés, pues te muestra un «esqueleto» de función útil para un control detallado de la lectura por teclado. Inspirándote en ella puedes escribir, por ejemplo, una función que sólo lea dígitos, o letras, o texto que satisface alguna determinada restricción.

5.2.4. Un par de utilidades

Hemos aprendido a crear ficheros y a modificar su contenido. No sabemos, sin embargo, cómo eliminar un fichero del sistema de ficheros ni cómo rebautizarlo. Hay dos funciones de la librería estándar de C (accesibles al incluir `stdio.h`) que permiten efectuar estas dos operaciones:

- *remove*: elimina el fichero cuya ruta se proporciona.

```
int remove(char ruta[]);
```

La función devuelve 0 si se consiguió eliminar el fichero y otro valor si se cometió algún error. ¡Ojo! No confundas borrar un fichero con borrar el contenido de un

> **La consulta de teclado**
>
> La función *getc* (o, para el caso, *fgetc* actuando sobre *stdin*) bloquea la ejecución del programa hasta que el usuario teclea algo y pulsa la tecla de retorno. Muchos programadores se preguntan ¿cómo puedo saber si una tecla está pulsada o no sin quedar bloqueado? Ciertas aplicaciones, como los videojuegos, necesitan efectuar consultas al estado del teclado no bloqueantes. Malas noticias: no es un asunto del lenguaje C, sino de bibliotecas específicas. El C estándar nada dice acerca de cómo efectuar esa operación.
>
> En Unix, la biblioteca *curses*, por ejemplo, permite manipular los terminales y acceder de diferentes modos al teclado. Pero no es una biblioteca fácil de (aprender a) usar. Y, además, presenta problemas de portabilidad, pues no necesariamente está disponible en todos los sistemas operativos.
>
> Cosa parecida podemos decir de otras cuestiones: sonido, gráficos tridimensionales, interfaces gráficas de usuario, etc. C, en tanto que lenguaje de programación estandarizado, no ofrece soporte. Eso sí: hay bibliotecas para infinidad de campos de aplicación. Tendrás que encontrar la que mejor se ajusta a tus necesidades y... ¡estudiar!

fichero. La función *remove* elimina completamente el fichero. Abrir un fichero en modo escritura y cerrarlo inmediatamente elimina su contenido, pero el fichero sigue existiendo (ocupando, eso sí, 0 bytes).

- *rename*: cambia el nombre de un fichero.

```
int rename(char ruta_original[], char nueva_ruta[]);
```

La función devuelve 0 si no hubo error, y otro valor en caso contrario.

5.3. Ficheros binarios

5.3.1. Abrir, leer/escribir, cerrar

La gestión de ficheros binarios obliga a trabajar con el mismo protocolo básico:

1. abrir el fichero en el modo adecuado,
2. leer y/o escribir información,
3. y cerrar el fichero.

La función de apertura de un fichero binario es la misma que hemos usado para los ficheros de texto: *fopen*. Lo que cambia es el modo de apertura: debe contener la letra `b`. Los modos de apertura básicos[3] para ficheros binarios son, pues:

- `"rb"` (lectura): El primer byte leído es el primero del fichero.
- `"wb"` (escritura): Trunca el fichero a longitud 0. Si el fichero no existe, se crea.
- `"ab"` (adición): Es un modo de escritura que preserva el contenido original del fichero. Los datos escritos se añaden al final del fichero.

Si el fichero no puede abrirse por cualquier razón, *fopen* devuelve el valor `NULL`.

La función de cierre del fichero es *fclose*.

Las funciones de lectura y escritura sí son diferentes:

[3] Más adelante te presentamos tres modos de apertura adicionales.

- *fread*: recibe una dirección de memoria, el número de bytes que ocupa un dato, el número de datos a leer y un fichero. He aquí su prototipo[4]:

```
int fread( void * direccion, int tam, int numdatos, FILE * fichero );
```

  Los bytes leídos se almacenan a partir de *direccion*. Devuelve el número de datos que ha conseguido leer (y si ese valor es menor que *numdatos*, es porque hemos llegado al final del fichero y no se ha podido efectuar la lectura completa).

- *fwrite*: recibe una dirección de memoria, el número de bytes que ocupa un dato, el número de datos a escribir y un fichero. Este es su prototipo:

```
int fwrite( void * direccion, int tam, int numdatos, FILE * fichero );
```

  Escribe en el fichero los *tam* por *numdatos* bytes existentes desde *direccion* en adelante. Devuelve el número de datos que ha conseguido escribir (si vale menos que *numdatos*, hubo algún error de escritura).

Empezaremos a comprender cómo trabajan estas funciones con un sencillo ejemplo. Vamos a escribir los diez primeros números enteros en un fichero:

diez_enteros.c

```
#include <stdio.h>

int main(void)
{
  FILE * fp;
  int i;

  fp = fopen("primeros.dat", "wb");
  for (i=0; i<10; i++)
    fwrite(&i, sizeof(int), 1, fp);
  fclose(fp);

  return 0;
}
```

Analicemos la llamada a *fwrite*. Fíjate: pasamos la dirección de memoria en la que empieza un entero (con &*i*) junto al tamaño en bytes de un entero (**sizeof**(**int**), que vale 4) y el valor 1. Estamos indicando que se van a escribir los 4 bytes (resultado de multiplicar 1 por 4) que empiezan en la dirección &*i*, es decir, se va a guardar en el fichero una copia exacta del contenido de *i*.

Quizá entiendas mejor qué ocurre con esta otra versión capaz de escribir un vector completo en una sola llamada a *fwrite*:

```
#include <stdio.h>

int main(void)
{
  FILE * fp;
  int i, v[10];

  for (i=0; i<10; i++)
    v[i] = i;
  fp = fopen("primeros.dat", "wb");
  fwrite(v, sizeof(int), 10, fp);
```

[4]Bueno, casi. El prototipo no usa el tipo **int**, sino **size_t**, que está definido como **unsigned int**. Preferimos presentarte una versión modificada del prototipo para evitar introducir nuevos conceptos.

```
    fclose(fp);

    return 0;
}
```

Ahora estamos pasando la dirección en la que empieza un vector (*v* es una dirección, así que no hemos de poner un & delante), el tamaño de un elemento del vector (**sizeof**(**int**)) y el número de elementos del vector (10). El efecto es que se escriben en el fichero los 40 bytes de memoria que empiezan donde empieza *v*. Resultado: todo el vector se almacena en disco con una sola operación de escritura. Cómodo, ¿no?

Ya te dijimos que la información de todo fichero binario ocupa exactamente el mismo número de bytes que ocuparía en memoria. Hagamos la prueba. Veamos con `ls -l`, desde el intérprete de comandos de Unix, cuánto ocupa el fichero:

```
$ ls -l primeros.dat ↲
-rw-r--r--    1 usuario  migrupo        40 may 10 11:00 primeros.dat
```

Efectivamente, ocupa exactamente 40 bytes (el número que aparece en quinto lugar). Si lo mostramos con `cat`, no sale nada con sentido en pantalla.

```
$ cat primeros.dat ↲
        $
```

¿Por qué? Porque `cat` interpreta el fichero como si fuera de texto, así que encuentra la siguiente secuencia binaria:

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000010
00000000 00000000 00000000 00000011
00000000 00000000 00000000 00000100
...
```

Los valores ASCII de cada grupo de 8 bits no siempre corresponden a caracteres visibles, por lo que no se representan como símbolos en pantalla (no obstante, algunos bytes sí tienen efecto en pantalla; por ejemplo, el valor 9 corresponde en ASCII al tabulador).

Hay una herramienta Unix que te permite inspeccionar un fichero binario: `od` (abreviatura de «octal dump», es decir, «volcado octal»).

```
$ od -l primeros.dat ↲
0000000          0          1          2          3
0000020          4          5          6          7
0000040          8          9
0000050
```

(La opción `-l` de `od` hace que muestre la interpretación como enteros de grupos de 4 bytes.) ¡Ahí están los números! La primera columna indica (en hexadecimal) el número de byte del primer elemento de la fila.

. . . . EJERCICIOS . . . .

▶ 325 ¿Qué aparecerá en pantalla si mostramos con el comando `cat` el contenido del fichero binario `otraprueba.dat` generado en este programa?:

otra_prueba.c — otra_prueba.c

```
#include <stdio.h>

int main(void)
{
```

```
  FILE * fp;
  int i, v[26];

  fp = fopen("otra_prueba.dat", "wb");
  for (i=97; i<123; i++)
    v[i-97] = i;
  fwrite(v, sizeof(int), 26, fp);
  fclose(fp);

  return 0;
}
```

(Una pista: el valor ASCII del carácter `'a'` es 97.)

¿Y qué aparecerá si lo visualizas con el comando `od -c` (la opción `-c` indica que se desea ver el fichero carácter a carácter e interpretado como secuencia de caracteres).

. . . .

Ya puedes imaginar cómo se leen datos de un fichero binario: pasando la dirección de memoria en la que queremos que se copie cierta cantidad de bytes del fichero. Los dos programas siguientes, por ejemplo, leen los diez valores escritos en los dos últimos programas. El primero lee entero a entero (de 4 bytes en 4 bytes), y el segundo con una sola operación de lectura (cargando los 40 bytes de golpe):

lee_primeros.c — **lee_primeros.c**

```
#include <stdio.h>

int main(void)
{
  FILE * fp;
  int i, n;

  fp = fopen("primeros.dat", "rb");
  for (i=0; i<10; i++) {
    fread(&n, sizeof(int), 1, fp);
    printf("%d\n", n);
  }
  fclose(fp);

  return 0;
}
```

lee_primeros2.c — **lee_primeros2.c**

```
#include <stdio.h>

int main(void)
{
  FILE * fd;
  int i, v[10];

  fp = fopen("primeros.dat", "rb");
  fread(v, sizeof(int), 10, fp);
  for (i=0; i<10; i++)
    printf("%d\n", v[i]);
  fclose(fp);

  return 0;
}
```

En los dos programas hemos indicado explícitamente que íbamos a leer 10 enteros, pues sabíamos de antemano que había exactamente 10 números en el fichero. Es fácil modificar el primer programa para que lea tantos enteros como haya, sin conocer a priori su número:

lee_todos.c   **lee_todos.c**

```c
#include <stdio.h>

int main(void)
{
  FILE * fp;
  int n;

  fp = fopen("primeros.dat", "rb");
  fread(&n, sizeof(int), 1, fp);
  while (!feof(fp)) {
    printf("%d\n", n);
    fread(&n, sizeof(int), 1, fp);
  }
  fclose(fp);

  return 0;
}
```

Lo cierto es que hay una forma más idiomática, más común en C de expresar lo mismo:

lee_todos2.c   **lee_todos2.c**

```c
#include <stdio.h>

int main(void)
{
  FILE * fp;
  int n;

  f = fopen("primeros.dat", "rb");
  while ( fread(&n, sizeof(int), 1, fp) == 1 )
    printf("%d\n", n);
  fclose(fp);

  return 0;
}
```

En esta última versión, la lectura de cada entero se efectúa con una llamada a *fread* en la condición del **while**.

. . . . EJERCICIOS . . . .

▶ 326 Diseña un programa que genere un fichero binario `primos.dat` con los 1000 primeros números primos.

▶ 327 Diseña un programa que añada al fichero binario `primos.dat` (ver ejercicio anterior) los 100 siguientes números primos. El programa leerá el contenido actual del fichero para averiguar cuál es el último primo conocido. A continuación, abrirá el fichero en modo adición y añadirá 100 nuevos primos. Si ejecutásemos dos veces el programa, el fichero acabaría conteniendo los 1200 primeros primos.

. . . .

No sólo puedes guardar tipos relativamente elementales. También puedes almacenar en disco tipos de datos creados por ti. Este programa, por ejemplo, lee de disco un vector de puntos, lo modifica y escribe en el fichero el contenido del vector:

escribe_registro.c escribe_registro.c

```
#include <stdio.h>
#include <math.h>

struct Punto {
   float x;
   float y;
};

int main(void)
{
   FILE * fp;
   struct Punto v[10];
   int i;

   // Cargamos en memoria un vector de puntos.
   fp = fopen("puntos.dat", "rb");
   fread(v, sizeof(struct Punto), 10, fp);
   fclose(fp);

   // Procesamos los puntos (calculamos el valor absoluto de cada coordenada).
   for (i=0; i<10; i++) {
      v[i].x = fabs(v[i].x);
      v[i].y = fabs(v[i].y);
   }

   // Escribimos el resultado en otro fichero.
   fp = fopen("puntos2.dat", "wb");
   fwrite(v, sizeof(struct Punto), 10, fp);
   fclose(fp);

   return 0;
}
```

Esta otra versión no carga el contenido del primer fichero completamente en memoria en una primera fase, sino que va leyendo, procesando y escribiendo punto a punto:

```
#include <stdio.h>
#include <math.h>

struct Punto {
   float x;
   float y;
};

int main(void)
{
   FILE * fp_entrada, * fp_salida;
   struct Punto p;
   int i;

   fp_entrada = fopen("puntos.dat", "rb");
   fp_salida = fopen("puntos2.dat", "wb");

   for (i=0; i<10; i++) {
      fread(&p, sizeof(struct Punto), 1, fp_entrada);
      p.x = fabs(p.x);
      p.y = fabs(p.y);
      fwrite(&p, sizeof(struct Punto), 1, fp_salida);
```

```
    }
    fclose(fp_entrada);
    fclose(fp_salida);

    return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 328 Los dos programas anteriores suponen que hay diez puntos en `puntos.dat`. Modifícalos para que procesen tantos puntos como haya en el fichero.

▶ 329 Implementa un programa que genere un fichero llamado `puntos.dat` con 10 elementos del tipo **struct** *Punto*. Las coordenadas de cada punto se generarán aleatoriamente en el rango $[-10, 10]$. Usa el último programa para generar el fichero `puntos2.dat`. Comprueba que contiene el valor absoluto de los valores de `puntos.dat`. Si es necesario, diseña un nuevo programa que muestre por pantalla el contenido de un fichero de puntos cuyo nombre suministra por teclado el usuario.

. . . .

5.3.2. Acceso directo

Los ficheros binarios pueden utilizarse como «vectores en disco» y acceder directamente a cualquier elemento del mismo. Es decir, podemos abrir un fichero binario en modo «lectura-escritura» y, gracias a la capacidad de desplazarnos libremente por él, leer/escribir cualquier dato. Es como si dispusieras del control de avance rápido hacia adelante y hacia atrás de un reproductor/grabador de cintas magnetofónicas. Con él puedes ubicar el «cabezal» de lectura/escritura en cualquier punto de la cinta y pulsar el botón «play» para escuchar (leer) o el botón «record» para grabar (escribir).

Además de los modos de apertura de ficheros binarios que ya conoces, puedes usar tres modos de lectura/escritura adicionales:

- `"r+b"`: No se borra el contenido del fichero, que debe existir previamente. El «cabezal» de lectura/escritura se sitúa al principio del fichero.
- `"w+b"`: Si el fichero no existe, se crea, y si existe, se trunca el contenido a longitud cero. El «cabezal» de lectura/escritura se sitúa al principio del fichero.
- `"a+b"`: Si el fichero no existe, se crea. El «cabezal» de lectura/escritura se sitúa al final del fichero.

Para poder leer/escribir a voluntad en cualquier posición de un fichero abierto en alguno de los modos binarios necesitarás dos funciones auxiliares: una que te permita desplazarte a un punto arbitrario del fichero y otra que te permita preguntar en qué posición te encuentras en un instante dado. La primera de estas funciones es *fseek*, que desplaza el «cabezal» de lectura/escritura al byte que indiquemos.

```
int fseek(FILE * fp, int desplazamiento, int desde_donde);
```

El valor *desde_donde* se fija con una constante predefinida que proporciona una interpretación distinta a *desplazamiento*:

- `SEEK_SET`: el valor de *desplazamiento* es un valor absoluto a contar desde el principio del fichero. Por ejemplo, `fseek(fp, 3, SEEK_SET)` desplaza al cuarto byte del fichero *fp*. (La posición 0 corresponde al primer byte del fichero.)

- SEEK_CUR: el valor de *desplazamiento* es un valor relativo al lugar en que nos encontramos en un instante dado. Por ejemplo, si nos encontramos en el cuarto byte del fichero *fp*, la llamada *fseek*(*fp*, -2, SEEK_CUR) nos desplazará al segundo byte, y *fseek*(*fp*, 2, SEEK_CUR) al sexto.

- SEEK_END: el valor de *desplazamiento* es un valor absoluto a contar desde el final del fichero. Por ejemplo, *fseek*(*fp*, -1, SEEK_END) nos desplaza al último byte de *fp*: si a continuación leyésemos un valor, sería el del último byte del fichero. La llamada *fseek*(*fp*, 0, SEEK_END) nos situaría fuera del fichero (en el mismo punto en el que estamos si abrimos el fichero en modo de adición).

La función devuelve el valor 0 si tiene éxito, y un valor no nulo en caso contrario.

Has de tener siempre presente que los desplazamientos sobre el fichero se indican en bytes. Si hemos almacenado enteros de tipo **int** en un fichero binario, deberemos tener la precaución de que todos nuestros *fseek* tengan desplazamientos múltiplos de **sizeof**(**int**).

Este programa, por ejemplo, pone a cero todos los valores pares de un fichero binario de enteros:

anula_pares.c — anula_pares.c

```
#include <stdio.h>

int main(void)
{
   FILE * fp;
   int n, bytes_leidos, cero = 0;

   fp = fopen("fichero.dat", "r+b");
   while (fread(&n, sizeof(int), 1, fp) != 0) {
      if (n % 2 == 0) { // Si el último valor leído es par...
         fseek(fp, -sizeof(int), SEEK_CUR); // ... damos un paso atrás ...
         fwrite(&cero, sizeof(int), 1, fp); // ... y sobreescribimos su valor absoluto.
      }
   }
   fclose(fp);

   return 0;
}
```

La segunda función que te presentamos en este apartado es *ftell*. Este es su prototipo:

```
int ftell(FILE *fp);
```

El valor devuelto por la función es la posición en la que se encuentra el «cabezal» de lectura/escritura en el instante de la llamada.

Veamos un ejemplo. Este programa, por ejemplo, crea un fichero y nos dice el número de bytes del fichero:

cuenta_bytes.c — cuenta_bytes.c

```
#include <stdio.h>

int main(void)
{
   FILE * fp;
   int i, pos;

   fp = fopen("prueba.dat", "wb");
   for (i=0; i<10; i++)
      fwrite(&i, sizeof(int), 1, fp);
```

```
    fclose(fp);

    fp = fopen("prueba.dat", "rb");
    fseek(fp, 0, SEEK_END);
    pos = ftell(fp);
    printf("Tamaño␣del␣fichero:␣%d\n", pos);
    fclose(fp);

    return 0;
}
```

Fíjate bien en el truco que permite conocer el tamaño de un fichero: nos situamos al final del fichero con *fseek* indicando que queremos ir al «primer byte desde el final» (byte 0 con el modo `SEEK_END`) y averiguamos a continuación la posición en la que nos encontramos (valor devuelto por *ftell*).

. . . . EJERCICIOS . . . .

▶ 330 Diseña una función de nombre *rebobina* que recibe un *FILE* * y nos ubica al inicio del mismo.

▶ 331 Diseña una función que reciba un *FILE* * (ya abierto) y nos diga el número de bytes que ocupa. Al final, la función debe dejar el cursor de lectura/escritura en el mismo lugar en el que estaba cuando se la llamó.

▶ 332 Diseña un programa que calcule y muestre por pantalla el máximo y el mínimo de los valores de un fichero binario de enteros.

▶ 333 Diseña un programa que calcule el máximo de los enteros de un fichero binario y lo intercambie por el que ocupa la última posición.

▶ 334 Nos pasan un fichero binario `dobles.dat` con una cantidad indeterminada de números de tipo **float**. Sabemos, eso sí, que los números están ordenados de menor a mayor. Diseña un programa que pida al usuario un número y determine si está o no está en el fichero.

En una primera versión, implementa una búsqueda secuencial que se detenga tan pronto estés seguro de que el número buscado está o no. El programa, en su versión final, deberá efectuar la búsqueda dicotómicamente (en un capítulo anterior se ha explicado qué es una búsqueda dicotómica).

. . . .

Trabajar con ficheros binarios como si se tratara de vectores tiene ciertas ventajas, pero también inconvenientes. La ventaja más obvia es la capacidad de trabajar con cantidades ingentes de datos sin tener que cargarlas completamente en memoria. El inconveniente más serio es la enorme lentitud con que se pueden ejecutar entonces los programas. Ten en cuenta que desplazarse por un fichero con *fseek* obliga a ubicar el «cabezal» de lectura/escritura del disco duro, una operación que es intrínsecamente lenta por comportar operaciones mecánicas, y no sólo electrónicas.

Si en un fichero binario mezclas valores de varios tipos resultará difícil, cuando no imposible, utilizar sensatamente la función *fseek* para posicionarse en un punto arbitrario del fichero. Tenemos un problema similar cuando la información que guardamos en un fichero es de longitud intrínsecamente variable. Pongamos por caso que usamos un fichero binario para almacenar una lista de palabras. Cada palabra es de una longitud, así que no hay forma de saber a priori en qué byte del fichero empieza la n-ésima palabra de la lista. Un truco consiste en guardar cada palabra ocupando tanto espacio como la palabra más larga. Este programa, por ejemplo, pide palabras al usuario y las escribe en un fichero binario en el que todas las cadenas miden exactamente lo mismo (aunque la longitud de cada una de ellas sea diferente):

Ficheros binarios en Python

Python también permite trabajar con ficheros binarios. La apertura, lectura/escritura y cierre de ficheros se efectúa con las funciones y métodos de Python que ya conoces: *open*, *read*, *write* y *close*. Con *read* puedes leer un número cualquiera de caracteres (de bytes) en una cadena. Por ejemplo, *f*.*read*(4) lee 4 bytes del fichero *f* (previamente abierto con *open*). Si esos 4 bytes corresponden a un entero (en binario), la cadena contiene 4 caracteres que lo codifican (aunque no de forma que los podamos visualizar cómodamente). ¿Cómo asignamos a una variable el valor entero codificado en esa cadena? Python proporciona un módulo con funciones que permiten pasar de binario a «tipos Python» y viceversa: el módulo **struct**. Su función *unpack* «desempaqueta» información binaria de una cadena. Para «desempaquetar» un entero de una cadena almacenada en una variable llamada *enbinario* la llamamos así: *unpack*("i", *enbinario*). El primer parámetro desempeña la misma función que las cadenas de formato en *scanf*, sólo que usa un juego de marcas de formato diferentes (`i` para el equivalente a un **int**, `d` para **float**, `q` para **long long**, etc.. Consulta el manual del módulo **struct** para conocerlos.). Aquí tienes un ejemplo de uso: un programa que lee y muestra los valores de un fichero binario de enteros:

```
from struct import unpack
f = open("primeros.dat", "r")
while 1:
    c = f.read(4)
    if c == '': break
    v = unpack("i", c)
    print v[0]
f.close()
```

Fíjate en que el valor devuelto por *unpack* no es directamente el entero, sino una lista (en realidad una tupla), por lo que es necesario indexarla para acceder al valor que nos interesa. La razón de que devuelva una lista es que *unpack* puede desempaquetar varios valores a la vez. Por ejemplo, *unpack*("iid", *cadena*) desempaqueta dos enteros y un flotante de *cadena* (que debe tener al menos 16 bytes, claro está). Puedes asignar los valores devueltos a tres variables así: *a*, *b*, *c* = *unpack*("iid", *cadena*).

Hemos aprendido, pues, a leer ficheros binarios con Python. ¿Cómo los escribimos? Siguiendo un proceso inverso: empaquetando primero nuestros «valores Python» en cadenas que los codifican en binario mediante la función *pack* y escribiendolas con el método *write*. Este programa de ejemplo escribe un fichero binario con los números del 0 al 99:

```
from struct import pack
f = open("primeros.dat", "w")
for v in range(100):
    c = pack("i", v)
    f.write(c)
f.close()
```

Sólo queda que aprendas a implementar acceso directo a los ficheros binarios con Python. Tienes disponibles los modos de apertura `'r+'`, `'w+'` y `'a+'`. Además, el método *seek* permite desplazarse a un byte cualquiera del fichero y el método *tell* indica en qué posición del fichero nos encontramos.

guarda_palabras.c   guarda_palabras.c

```
#include <stdio.h>

#define MAXLON 80

```

```c
int main(void)
{
   char palabra[MAXLON+1], seguir[MAXLON+1];
   FILE * fp;

   fp = fopen("diccio.dat", "wb");
   do {
      printf("Dame una palabra: "); gets(palabra);
      fwrite(palabra, sizeof(char), MAXLON, fp);
      printf("Pulsa 's' para añadir otra."); gets(seguir);
   } while (strcmp(seguir, "s") == 0);
   fclose(fp);

   return 0;
}
```

Fíjate en que cada palabra ocupa siempre lo mismo, independientemente de su longitud: 80 bytes. Este otro programa es capaz ahora de mostrar la lista de palabras en orden inverso, gracias a la ocupación fija de cada palabra:

lee_palabras_orden_inverso.c lee_palabras_orden_inverso.c

```c
#include <stdio.h>

#define MAXLON 80

int main(void)
{
   FILE * fp;
   char palabra[MAXLON+1];
   int tam;

   /* primero, averiguar el tamaño del fichero (en palabras) */
   fp = fopen("diccio.dat", "rb");
   fseek(fp, 0, SEEK_END);
   tam = ftell(fp) / MAXLON;

   /* y ya podemos listarlas en orden inverso */
   for (i=tam-1; i>=0; i--) {
      fseek(fp, i * MAXLON, SEEK_SET);
      fread(palabra, sizeof(char), MAXLON, fp);
      printf("%s\n", palabra);
   }
   fclose(fp);

   return 0;
}
```

. . . . EJERCICIOS . . . .

▶ 335 Los dos programas anteriores pueden plantear problemas cuando trabajan con palabras que tienen 80 caracteres más el terminador. ¿Qué problemas? ¿Cómo los solucionarías?

▶ 336 Diseña un programa que lea una serie de valores enteros y los vaya escribiendo en un fichero hasta que el usuario introduzca el valor −1 (que no se escribirá en el fichero). Tu programa debe, a continuación, determinar si la secuencia de números introducida en el fichero es palíndroma.

▶ 337 Deseamos gestionar una colección de cómics. De cada cómic anotamos los siguientes datos:

- Superhéroe: una cadena de hasta 20 caracteres.
- Título: una cadena de hasta 200 caracteres.
- Número: un entero.
- Año: un entero.
- Editorial: una cadena de hasta 30 caracteres.
- Sinopsis: una cadena de hasta 1000 caracteres.

El programa permitirá:

1. Dar de alta un cómic.
2. Consultar la ficha completa de un cómic dado el superhéroe y el número del episodio.
3. Ver un listado por superhéroe que muestre el título de todas sus historias.
4. Ver un listado por año que muestre el superhérore y título de todas sus historias.

Diseña un programa que gestione la base de datos teniendo en cuenta que no queremos cargarla en memoria cada vez que ejecutamos el programa, sino gestionarla directamente sobre disco.

. . . .

Truncamiento de ficheros

Las funciones estándar de manejo de ficheros no permiten efectuar una operación que puede resultar necesaria en algunas aplicaciones: eliminar elementos de un fichero. Una forma de conseguir este efecto consiste en generar un nuevo fichero en el que escribimos sólo aquellos elementos que no deseamos eliminar. Una vez generado el nuevo fichero, borramos el original y renombramos el nuevo para que adopte el nombre del original. Costoso.

En Unix puedes recurrir a la función *truncate* (disponible al incluir la cabecera `unistd.h`). El perfil de *truncate* es éste:

```
int truncate(char nombre[], int longitud);
```

La función recibe el nombre de un fichero (que no debe estar abierto) y el número de bytes que deseamos conservar. Si la llamada tiene éxito, la función hace que en el fichero sólo permanezcan los *longitud* primeros bytes y devuelve el valor 0. En caso contrario, devuelve el valor −1. Observa que sólo puedes borrar los últimos elementos de un fichero, y no cualquiera de ellos. Por eso la acción de borrar parte de un fichero recibe el nombre de truncamiento.

5.4. Errores

Algunas de las operaciones con ficheros pueden resultar fallidas (apertura de un fichero cuya ruta no apunta a ningún fichero existente, cierre de un fichero ya cerrado, etc.). Cuando así ocurre, la función llamada devuelve un valor que indica que se cometió un error, pero ese valor sólo no aporta información que nos permita conocer el error cometido.

La información adicional está codificada en una variable especial: *errno* (declarada en `errno.h`). Puedes comparar su valor con el de las constantes predefinidas en `errno.h` para averiguar qué error concreto se ha cometido:

- `EACCESS`: permiso denegado,
- `EEXIST`: el fichero no existe,
- `EMFILE`: demasiados ficheros abiertos,
- ...

Como manejarte con tantas constantes (algunas con significados un tanto difícil de comprender hasta que curses asignaturas de sistemas operativos) resulta complicado, puedes usar una función especial:

```
void perror (char s[]);
```

Esta función muestra por pantalla el valor de la cadena *s*, dos puntos y un mensaje de error que detalla la causa del error cometido. La cadena *s*, que suministra el programador, suele indicar el nombre de la función en la que se detectó el error, ayudando así a la depuración del programa.

Apéndice A

Tipos básicos

A.1. Enteros

A.1.1. Tipos

Esta tabla muestra el nombre de cada uno de los tipos de datos para valores enteros (algunos tienen dos nombres válidos), su rango de representación y el número de bytes (grupos de 8 bits) que ocupan.

Tipo	Rango	Bytes
char	$-128 \ldots 127$	1
short int (o **short**)	$-32768 \ldots 32767$	2
int	$-2147483648 \ldots 2147483647$	4
long int (o **long**)	$-2147483648 \ldots 2147483647$	4
long long int (o **long long**)	$-9223372036854775808 \ldots 9223372036854775807$	8

(Como ves, los tipos **short int**, **long int** y **long long int** pueden abreviarse, respectivamente, como **short**, **long**, y **long long**.)

Un par de curiosidades sobre la tabla de tipos enteros:

- Los tipos **int** y **long int** ocupan lo mismo (4 bytes) y tienen el mismo rango. Esto es así *para el compilador* gcc *sobre un PC*. En una máquina distinta o con otro compilador, podrían ser diferentes: los **int** podrían ocupar 4 bytes y los **long int**, 8, por ejemplo. En sistemas más antiguos un **int** ocupaba 2 bytes y un **long int**, 4.

- El nombre del tipo **char** es abreviatura de «carácter» («character», en inglés) y, sin embargo, hace referencia a los enteros de 8 bits, es decir, 1 byte. Los valores de tipo **char** son ambivalentes: son tanto números enteros como caracteres.

Es posible trabajar con enteros sin signo en C, es decir, números enteros positivos. La ventaja de trabajar con ellos es que se puede aprovechar el bit de signo para aumentar el rango positivo y duplicarlo. Los tipos enteros sin signo tienen el mismo nombre que sus correspondientes tipos con signo, pero precedidos por la palabra **unsigned**, que actúa como un adjetivo:

Tipo	Rango	Bytes
unsigned char	$0 \ldots 255$	1
unsigned short int (o **unsigned short**)	$0 \ldots 65535$	2
unsigned int (o **unsigned**)	$0 \ldots 4294967295$	4
unsigned long int (o **unsigned long**)	$0 \ldots 4294967295$	4
unsigned long long int (o **unsigned long long**)	$0 \ldots 18446744073709551615$	8

Del mismo modo que podemos «marcar» un tipo entero como «sin signo» con el adjetivo **unsigned**, podemos hacer explícito que tiene signo con el adjetivo **signed**. O sea, el tipo **int** puede escribirse también como **signed int**: son exactamente el mismo tipo, sólo que en el segundo caso se pone énfasis en que tiene signo, haciendo posible una mejora en la legibilidad de un programa donde este rasgo sea importante.

A.1.2. Literales

Puedes escribir números enteros en notación octal (base 8) o hexadecimal (base 16). Un número en notación hexadecimal empieza por 0*x*. Por ejemplo, 0*xff* es 255 y 0*x*0 es 0. Un número en notación octal debe empezar por un 0 y *no* ir seguido de una *x*. Por ejemplo, 077 es 63 y 010 es 8.[1]

Puedes precisar que un número entero es largo añadiéndole el sufijo `L` (por «Long»). Por ejemplo, 2L es el valor 2 codificado con 32 bits. El sufijo `LL` (por «long long») indica que el número es un **long long int**. El literal 2LL, por ejemplo, representa al número entero 2 codificado con 64 bits (lo que ocupa un **long long int**). El sufijo `U` (combinado opcionalmente con `L` o `LL`) precisa que un número no tiene signo (la `U` por «unsigned»).

Normalmente no necesitarás usar esos sufijos, pues C hace conversiones automáticas de tipo cuando conviene. Sí te hará falta si quieres denotar un número mayor que 2147483647 (o menor que −2147483648), pues en tal caso el número no puede representarse como un simple **int**. Por ejemplo, la forma correcta de referirse a 3000000000 es con el literal 3000000000LL.

C resulta abrumador por la gran cantidad de posibilidades que ofrece. Son muchas formas diferentes de representar enteros, ¿verdad? No te preocupes, sólo en aplicaciones muy concretas necesitarás utilizar la notación octal o hexadecimal o tendrás que añadir el sufijo a un literal para indicar su tipo.

A.1.3. Marcas de formato

Hay una marca de formato para la impresión o lectura de valores de cada tipo de entero:

Tipo	Marca	Tipo	Marca
char (número)	`%hhd`	**unsigned char**	`%hhu`
short	`%hd`	**unsigned short**	`%hu`
int	`%d`	**unsigned**	`%u`
long	`%ld`	**unsigned long**	`%lu`
long long	`%lld`	**unsigned long long**	`%llu`

Puedes mostrar los valores numéricos en base octal o hexadecimal sustituyendo la `d` (o la `u`) por una `o` o una `x`, respectivamente. Por ejemplo, `%lx` es una marca que muestra un entero largo en hexadecimal y `%ho` muestra un **short** en octal.

Son muchas, ¿verdad? La que usarás más frecuentemente es `%d`. De todos modos, por si necesitas utilizar otras, he aquí algunas reglas mnemotécnicas:

- `d` significa «decimal» y alude a la base en que se representa la información: base 10. Por otra parte, `x` y `o` representan a «hexadecimal» y «octal» y aluden a las bases 16 y 8.

- `u` significa «unsigned», es decir, «sin signo».

[1]Lo cierto es que también puede usar notación octal o hexadecimal en Python, aunque en su momento no lo contamos.

- `h` significa «mitad» (por «half»), así que `%hd` es «la mitad» de un entero, o sea, un **short**, y `%hhd` es «la mitad de la mitad» de un entero, o sea, un **char**.
- `l` significa «largo» (por «long»), así que `%ld` es un entero largo (un **long**) y `%lld` es un entero extra-largo (un **long long**).

A.2. Flotantes

A.2.1. Tipos

También en el caso de los flotantes tenemos dónde elegir: hay tres tipos diferentes. En esta tabla te mostramos el nombre, máximo valor absoluto y número de *bits* de cada uno de ellos:

Tipo	Máximo valor absoluto	Bytes
float	$3.40282347 \cdot 10^{38}$	4
double	$1.7976931348623157 \cdot 10^{308}$	8
long double	$1.189731495357231765021263853031 \cdot 10^{4932}$	12

Recuerda que los números expresados en coma flotante presentan mayor resolución en la cercanías del 0, y que ésta es tanto menor cuanto mayor es, en valor absoluto, el número representado. El número no nulo más próximo a cero que puede representarse con cada uno de los tipos se muestra en esta tabla:

Tipo	Mínimo valor absoluto no nulo
float	$1.17549435 \cdot 10^{-38}$
double	$2.2250738585072014 \cdot 10^{-308}$
long double	$3.3621031431120935062626778173218 \cdot 10^{-4932}$

A.2.2. Literales

Ya conoces las reglas para formar literales para valores de tipo **float**. Puedes añadir el sufijo `F` para precisar que el literal corresponde a un **double** y el sufijo `L` para indicar que se trata de un **long double**. Por ejemplo, el literal 3.2`F` es el valor 3.2 codificado como **double**. Al igual que con los enteros, normalmente no necesitarás precisar el tipo del literal con el sufijo `L`, a menos que su valor exceda del rango propio de los **float**.

A.2.3. Marcas de formato

Veamos ahora las principales marcas de formato para la **impresión de datos de tipos flotantes**:

Tipo	Notación convencional	Notación científica
float	`%f`	`%e`
double	`%f`	`%e`
long double	`%Lf`	`%Le`

Observa que tanto **float** como **double** usan la misma marca de formato *para impresión* (o sea, con la función *printf* y similares).

No pretendemos detallar todas las marcas de formato para flotantes. Tenemos, además, otras como `%E`, `%F`, `%g`, `%G`, `%LE`, `%LF`, `%Lg` y `%LG`. Cada marca introduce ciertos matices que, en según qué aplicaciones, pueden venir muy bien. Necesitarás un buen manual

de referencia a mano para controlar estos y otros muchos aspectos (no tiene sentido memorizarlos) cuando ejerzas de programador en C durante tu vida profesional.[2]

Las marcas de formato para la **lectura de datos de tipos flotantes** presentan alguna diferencia:

Tipo	Notación convencional
float	`%f`
double	`%lf`
long double	`%Lf`

Observa que la marca de impresión de un **double** es `%f`, pero la de lectura es `%lf`. Es una incoherencia de C que puede darte algún que otro problema.

A.3. Caracteres

El tipo **char**, que ya hemos presentado al estudiar los tipos enteros, es, a la vez el tipo con el que solemos representar caracteres y con el que formamos las cadenas.

A.3.1. Literales

Los literales de carácter encierran entre comillas simples al carácter en cuestión o lo codifican como un número entero. Es posible utilizar secuencias de escape para indicar el carácter que se encierra entre comillas.

A.3.2. Marcas de formato

Los valores de tipo **char** pueden mostrarse en pantalla (o escribirse en ficheros de texto) usando la marca `%c` o `%hhd`. La primera marca muestra el carácter como eso mismo, como carácter; la segunda muestra su valor decimal (el código ASCII del carácter).

A.4. Otros tipos básicos

C99 define tres nuevos tipos básicos: el tipo lógico (o booleano), el tipo complejo y el tipo imaginario.

A.4.1. El tipo booleano

Las variables de tipo **_Bool** pueden almacenar los valores 0 («falso») o 1 («cierto»). Si se incluye la cabecera `<stdbool.h>` es posible usar el identificador de tipo **bool** y las constantes `true` y `false` para referirse al tipo **_Bool** y a los valores 1 y 0, respectivamente.

A.4.2. Los tipos complejo e imaginario

C99 ofrece soporte para la aritmética compleja a través de los tipos **_Complex** e **_Imaginary**.

[2]En Unix puedes obtener ayuda acerca de las funciones estándar con el manual en línea. Ejecuta `man 3 printf`, por ejemplo, y obtendrás una página de manual sobre la función *printf*, incluyendo información sobre todas sus marcas de formato y modificadores.

A.5. Una reflexión acerca de la diversidad de tipos escalares

¿Por qué ofrece C tan gran variedad de tipos de datos para enteros y flotantes? Porque C procura facilitar el diseño de programas eficientes proporcionando al programador un juego de tipos que le permita adoptar el compromiso adecuado entre ocupación de memoria y rango disponible. ¿Por qué iba un programador a querer gastar 4 bytes en una variable que sólo almacenará valores entre 0 y 255? Naturalmente, ofrecer más control no es gratis: a cambio hemos de tomar muchas más decisiones. Ahorrar 3 bytes en una variable puede no justificar el quebradero de cabeza, pero piensa en el ahorro que se puede producir en un vector que contiene miles o cientos de miles de elementos que pueden representarse cada uno con un **char** en lugar de con un **int**.

Por otra parte, la arquitectura de tu ordenador está optimizada para realizar cálculos con valores de ciertos tamaños. Por ejemplo, las operaciones con enteros suelen ser más rápidas si trabajas con **int** (aunque ocupen más bytes que los **char** o **short**) y las operaciones con flotantes más eficientes trabajan con **double**.

Según si valoras más velocidad o consumo de memoria en una aplicación, deberás escoger uno u otro tipo de datos para ciertas variables.

Apéndice B

La lectura de datos por teclado, paso a paso

B.1. La lectura de valores escalares con *scanf*

La función *scanf* (y *fscanf*) se comporta de un modo un tanto especial y puede desconcertarte en ocasiones. Veamos qué hace exactamente *scanf*:

- Empieza saltándose los blancos que encuentra (espacios en blanco, tabuladores y saltos de línea).
- A continuación, «consume» los caracteres no blancos mientra «le sirvan» para leer un valor del tipo que se indica con la marca de formato (por ejemplo, dígitos si la marca es %d).
- La lectura se detiene cuando el siguiente carácter a leer «no sirve» (por ejemplo, una letra si estamos leyendo un entero). Dicho carácter no es «consumido». Los caracteres «consumidos» hasta este punto se interpretan como la representación de un valor del tipo que se indica con la correspondiente marca de formato, así que se crea dicho valor y se escribe en la zona de memoria que empieza en la dirección que se indique.

Un ejemplo ayudará a entender el comportamiento de *scanf*:

lee_tres.c

```
#include <stdio.h>

int main(void)
{
  int a, c;
  float b;

  printf("Entero a: "); scanf("%d", &a);
  printf("Flotante b: "); scanf("%f", &b);
  printf("Entero c: "); scanf("%d", &c);
  printf ("El entero a es %d, el flotante b es %f
", a, b);
  printf ("y el entero c es %d
", c);

  return 0;
}
```

Ejecutemos el programa e introduzcamos los valores 20, 3.0 y 4 pulsando el retorno de carro tras cada uno de ellos.

```
Entero a: 20
Flotante b: 3.0
Entero c: 4
El entero a es 20, el flotante b es 3.000000
y el entero c es 4
```

Perfecto. Para ver qué ha ocurrido paso a paso vamos a representar el texto que escribe el usuario durante la ejecución como una secuencia de teclas. En este gráfico se muestra qué ocurre durante la ejecución del primer *scanf* (línea 8), momento en el que las tres variables están sin inicializar y el usuario acaba de pulsar las teclas 2, 0 y retorno de carro:



El carácter a la derecha de la flecha es el siguiente carácter que va a ser consumido.

La ejecución del primer *scanf* consume los caracteres '2' y '0', pues ambos son válidos para formar un entero. La función detecta el blanco (salto de línea) que sigue al carácter '0' y se detiene. Interpreta entonces los caracteres que ha leído como el valor entero 20 y lo almacena en la dirección de memoria que se la suministrado (&*a*):

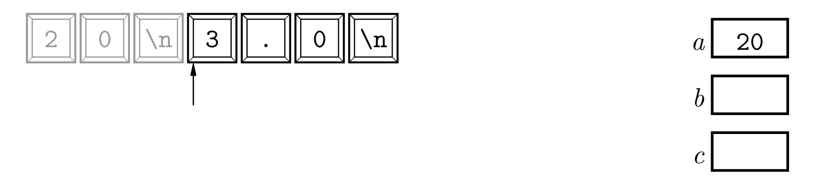


En la figura hemos representado los caracteres consumidos en color gris. Fíjate en que el salto de línea aún no ha sido consumido.

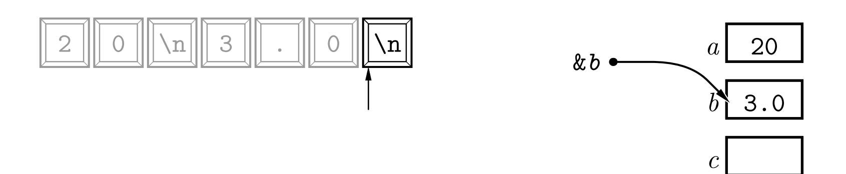
La ejecución del segundo *scanf*, el que lee el contenido de *b*, empieza descartando los blancos iniciales, es decir, el salto de línea:



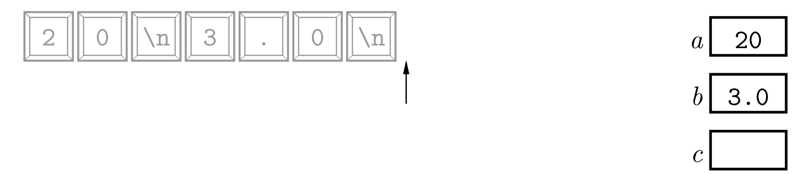
Como no hay más caracteres que procesar, *scanf* queda a la espera de que el usuario teclee algo con lo que pueda formar un flotante y pulse retorno de carro. Cuando el usaurio teclea el 3.0 seguido del salto de línea, pasamos a esta nueva situación:



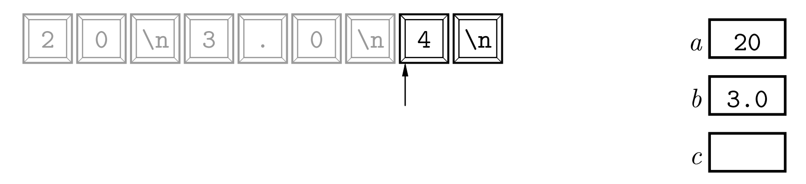
Ahora, *scanf* reanuda su ejecución y consume el '3', el '.' y el '0'. Como detecta que lo que sigue no es válido para formar un flotante, se detiene, interpreta los caracteres leídos como el valor flotante 3.0 y lo almacena en la dirección de *b*:



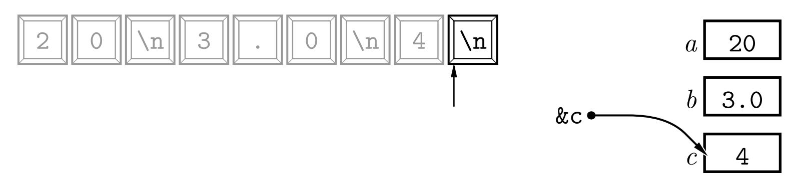
Finalmente, el tercer *scanf* entra en ejecución y empieza por saltarse el salto de línea.



Acto seguido se detiene, pues no es necesario que el usuario introduzca nuevo texto que procesar. Entonces el usuario escribe el 4 y pulsa retorno:



Ahora *scanf* prosigue consumiendo el 4 y deteniéndose nuevamente ante el salto de línea. El carácter leído se interpreta entonces como el entero 4 y se almacena en la dirección de memoria de *c*:



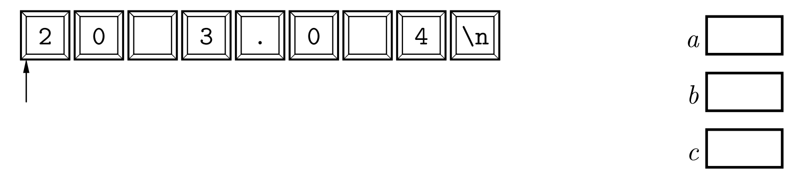
Como puedes apreciar, el último salto de línea no llega a ser consumido, pero eso importa poco, pues el programa finaliza correctamente su ejecución.

Vamos a estudiar ahora el porqué de un efecto curioso. Imagina que, cuando el programa pide al usuario el primer valor entero, éste introduce tanto dicho valor como los dos siguientes, separando los tres valores con espacios en blanco. He aquí el resultado en pantalla:

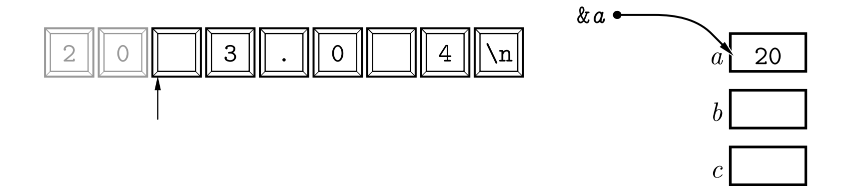
```
Entero a: 20 3.0 4
Flotante b: Entero c: El entero a es 20, el flotante b es 3.000000 y el entero c es 4
```

El programa ha leído correctamente los tres valores, *sin esperar a que el usuario introduzca tres líneas con datos*: cuando tenía que detenerse para leer el valor de *b*, no lo ha hecho, pues «sabía» que ese valor era 3.0; y tampoco se ha detenido al leer el valor de *c*, ya que de algún modo «sabía» que era 4. Veamos paso a paso lo que ha sucedido, pues la explicación es bien sencilla.

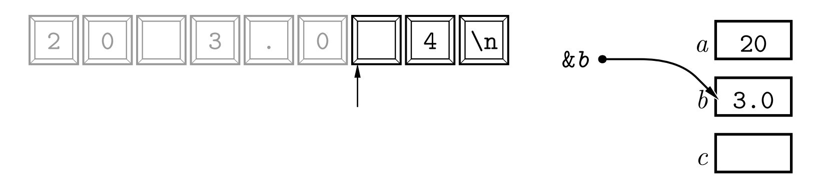
Durante la ejecución del primer *scanf*, el usuario ha escrito el siguiente texto:



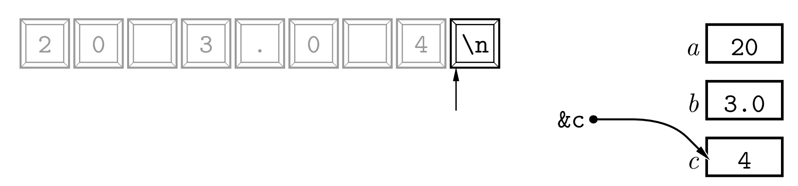
Como su objetivo es leer un entero, ha empezado a consumir caracteres. El '2' y el '0' le son últiles, así que los ha consumido. Entonces se ha detenido frente al espacio en blanco. Los caracteres leídos se interpretan como el valor entero 20 y se almacenan en *a*:



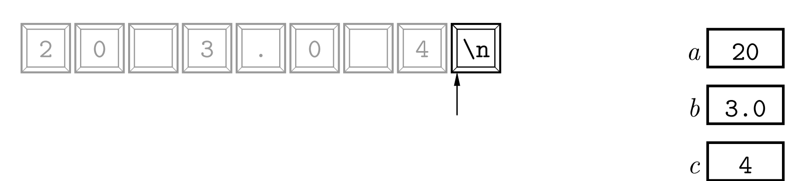
La ejecución del siguiente *scanf* no ha detenido la ejecución del programa, pues aún había caracteres pendientes de procesar en la entrada. Como siempre, *scanf* se ha saltado el primer blanco y ha ido encontrando caracteres válidos para ir formando un valor del tipo que se le indica (en este caso, un flotante). La función *scanf* ha dejado de consumir caracteres al encontrar un nuevo blanco, se ha detenido y ha almacenado en *b* el valor flotante `3.0`. He aquí el nuevo estado:



Finalmente, el tercer *scanf* tampoco ha esperado nueva entrada de teclado: se ha saltado directamente el siguiente blanco, ha encontrado el carácter `'4'` y se ha detenido porque el carácter `\n` que le sigue es un blanco. El valor leído (el entero 4) se almacena en c:



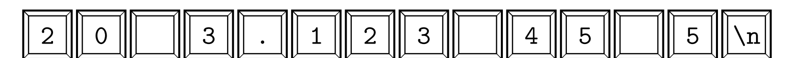
Tras almacenar en `c` el entero `4`, el estado es éste:



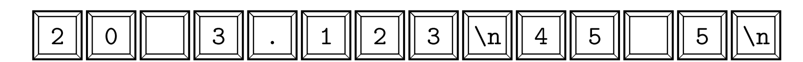
Cuando observes un comportamiento inesperado de *scanf*, haz un análisis de lo sucedido como el que te hemos presentado aquí y verás que todo tiene explicación.

. . . . EJERCICIOS . . . .

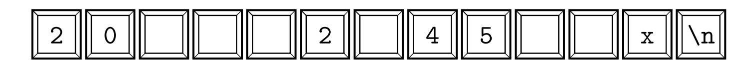
▶ 338 ¿Qué pasa si el usuario escribe la siguiente secuencia de caracteres como datos de entrada en la ejecución del programa?



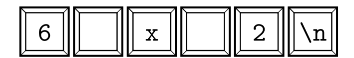
▶ 339 ¿Qué pasa si el usuario escribe la siguiente secuencia de caracteres como datos de entrada en la ejecución del programa?



▶ 340 ¿Qué pasa si el usuario escribe la siguiente secuencia de caracteres como datos de entrada en la ejecución del programa?



▶ 341 ¿Qué pasa si el usuario escribe la siguiente secuencia de caracteres como datos de entrada en la ejecución del programa?



(Prueba este ejercicio con el ordenador.)

B.2. La lectura de cadenas con *scanf*

Vamos a estudiar ahora el comportamiento paso a paso de *scanf* cuando leemos una cadena:

- Se descartan los blancos iniciales (espacios en blanco, tabuladores o saltos de línea).
- Se leen los caracteres «válidos» *hasta el primer blanco* y se almacenan en posiciones de memoria consecutivas a partir de la que se suministra como argumento. Se entiende por carácter válido cualquier carácter no blanco (ni tabulador, ni espacio en blanco, ni salto de línea...).
- Se añade al final un terminador de cadena.

Un ejemplo ayudará a entender qué ocurre ante ciertas entradas:

lee_cadena.c

```
#include <stdio.h>

#define TALLA 10

int main(void)
{
  char a[TALLA+1], b[TALLA+1];

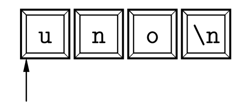
  printf("Cadena␣1:␣"); scanf("%s", a);
  printf("Cadena␣2:␣"); scanf("%s", b);
  printf ("La␣cadena␣1␣es␣%s␣y␣la␣cadena␣2␣es␣%s\n", a, b);

  return 0;
}
```

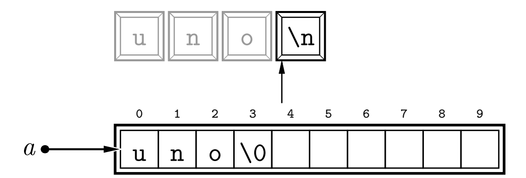
Si ejecutas el programa y escribes una primera cadena sin blancos, pulsas el retorno de carro, escribes otra cadena sin blancos y vuelves a pulsar el retorno, la lectura se efectúa como cabe esperar:

```
Cadena 1: uno
Cadena 2: dos
La cadena 1 es uno y la cadena 2 es dos
```

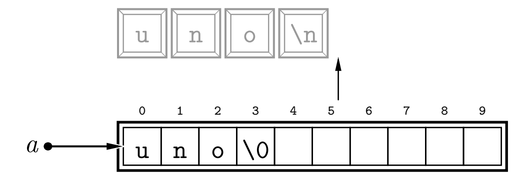
Estudiemos paso a paso lo ocurrido. Ante el primer *scanf*, el usuario ha escrito lo siguiente:



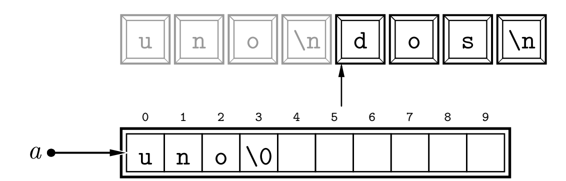
La función ha empezado a consumir los caracteres con los que ir formando la cadena. Al llegar al salto de línea se ha detenido sin consumirlo. He aquí el nuevo estado de cosas:



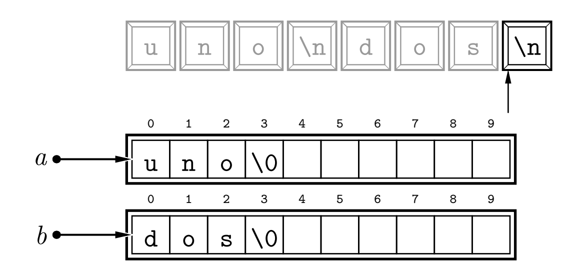
(Fíjate en que *scanf* termina correctamente la cadena almacenada en *a*.) Acto seguido se ha ejecutado el segundo *scanf*. La función se salta entonces el blanco inicial, es decir, el salto de línea que aún no había sido consumido.



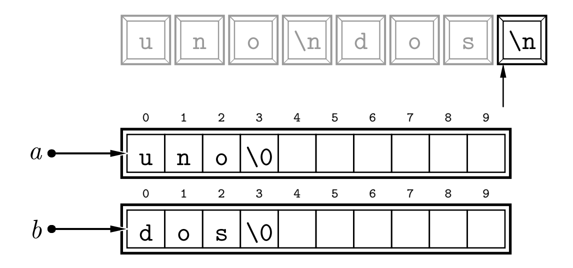
Como no hay más caracteres, *scanf* ha detenido la ejecución a la espera de que el usuario teclee algo. Entonces el usuario ha escrito la palabra `dos` y ha pulsado retorno de carro:



Entonces *scanf* ha procedido a consumir los tres primeros caracteres:



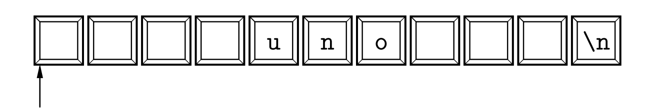
Fíjate en que *scanf* introduce automáticamente el terminador pertinente al final de la cadena leída. El segundo *scanf* nos conduce a esta nueva situación:



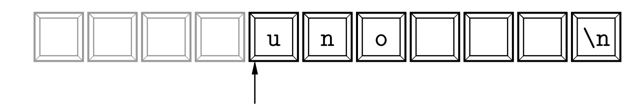
Compliquemos un poco la situación. ¿Qué ocurre si, al introducir las cadenas, metemos espacios en blanco delante y detrás de las palabras?

```
Cadena 1: ␣␣␣␣uno␣␣
Cadena 2: ␣␣dos␣
La cadena 1 es uno y la cadena 2 es dos
```

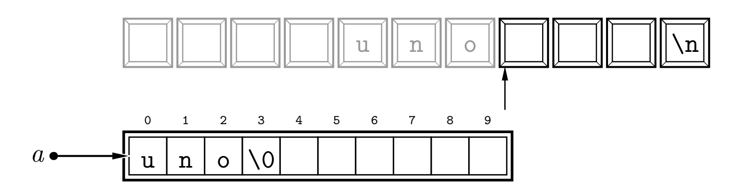
Recuerda que *scanf* se salta siempre los blancos que encuentra al principio y que se detiene en el primer espacio que encuentra tras empezar a consumir caracteres válidos. Veámoslo paso a paso. Empezamos con este estado de la entrada:



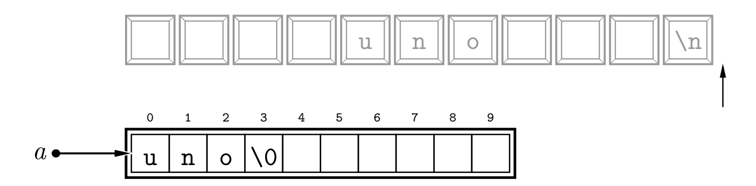
El primer *scanf* empieza saltándose los blancos inciales:



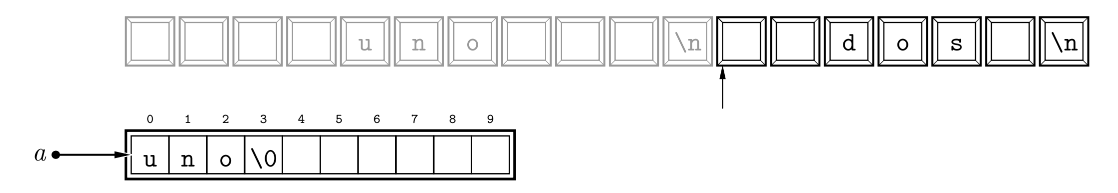
A continuación consume los caracteres 'u', 'n' y 'o' y se detiene al detectar el blanco que sigue:



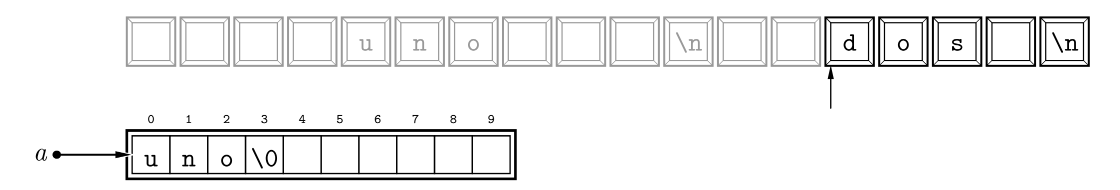
Cuando se ejecuta, el segundo *scanf* empieza saltándose los blancos iniciales, que son todos los que hay hasta el salto de línea (incluído éste):



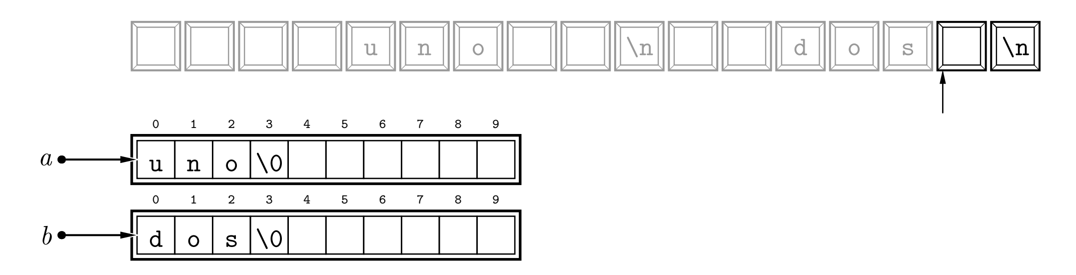
De nuevo, como no hay más que leer, la ejecución se detiene. El usuario teclea entonces nuevos caracteres:



A continuación, sigue saltándose los blancos:



Pasa entonces a consumir caracteres no blancos y se detiene ante el primer blanco:

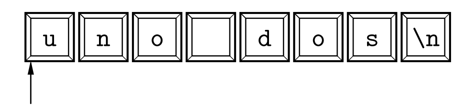


Ya está.

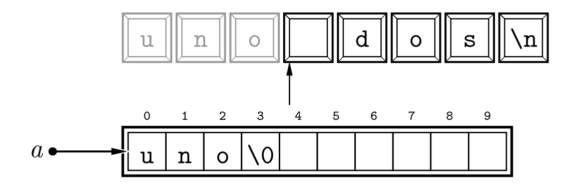
Imagina ahora que nuestro usuario quiere introducir en *a* la cadena `"uno␣dos"` y en *b* la cadena `"tres"`. Aquí tienes lo que ocurre al ejecutar el programa

```
Cadena 1: uno dos
Cadena 2: La cadena 1 es uno y la cadena 2 es dos
```

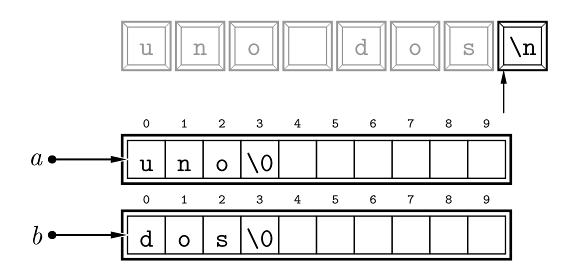
El programa ha finalizado sin darle tiempo al usuario a introducir la cadena `"tres"`. Es más, la primera cadena vale `"uno"` y la segunda `"dos"`, con lo que ni siquiera se ha conseguido el primer objetivo: leer la cadena `"uno␣dos"` y depositarla tal cual en *a*. Analicemos paso a paso lo sucedido. La entrada que el usuario teclea ante el primer *scanf* es ésta:



La función lee en *a* los caracteres `'u'`, `'n'` y `'o'` y se detiene al detectar un blanco. El nuevo estado se puede representar así:



El segundo *scanf* entra en juego entonces y «aprovecha» lo que aún no ha sido procesado, así que empieza por descartar el blanco inicial y, a continuación, consume los caracteres `'d'`, `'o'`, `'s'`:



¿Ves? La consecuencia de este comportamiento es que con *scanf sólo podemos leer palabras individuales*. Para leer una línea completa en una cadena, hemos de utilizar una función distinta: *gets* (por «get string», que en inglés significa «obtén cadena»), disponible incluyendo `stdio.h` en nuestro programa.

B.3. Un problema serio: la lectura alterna de cadenas con *gets* y de escalares con *scanf*

Vamos a estudiar un caso concreto y analizaremos las causas del extraño comportamiento observado.

lee_alterno_mal.c lee_alterno_mal.c

```
#include <stdio.h>

#define TALLA 80

int main(void)
{
```

```
    char a[TALLA+1], b[TALLA+1];
    int i;

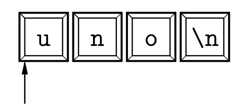
    printf("Cadena␣a:␣"); gets(a);
    printf("Entero␣i:␣"); scanf("%d", &i);
    printf("Cadena␣b:␣"); gets(b);
    printf ("La␣cadena␣a␣es␣%s,␣el␣entero␣i␣es␣%d␣y␣la␣cadena␣b␣es␣%s\n",
            a, i, b);

    return 0;
}
```

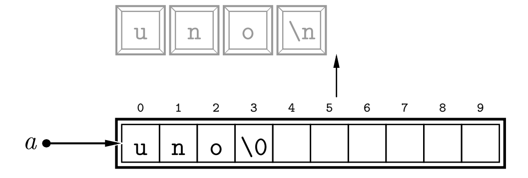
Observa que leemos cadenas con *gets* y un entero con *scanf*. Vamos a ejecutar el programa introduciendo la palabra `uno` en la primera cadena, el valor `2` en el entero y la palabra `dos` en la segunda cadena.

```
Cadena a: uno
Entero i: 2
Cadena b: La cadena a es uno, el entero i es 2 y la cadena b es
```

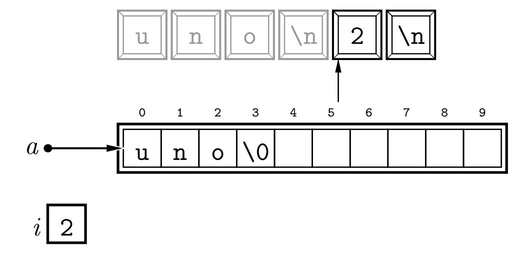
¿Qué ha pasado? No hemos podido introducir la segunda cadena: ¡tan pronto hemos escrito el retorno de carro que sigue al 2, el programa ha finalizado! Estudiemos paso a paso lo ocurrido. El texto introducido ante el primer *scanf* es:



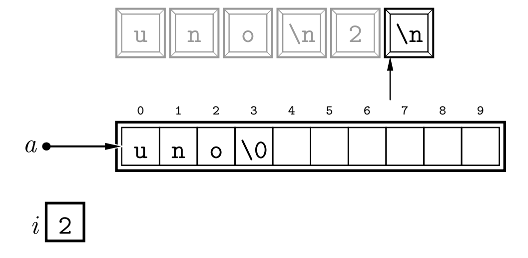
El primer *gets* nos deja en esta situación:



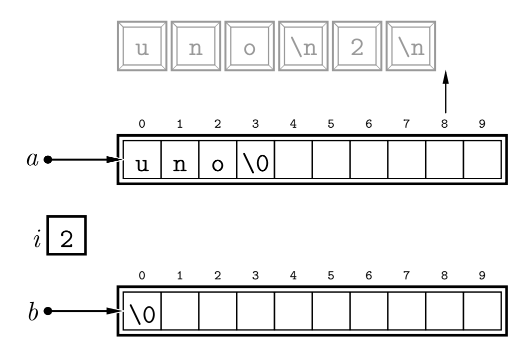
A continuación se ejecuta el *scanf* con el que se lee el valor de *i*. El usuario teclea lo siguiente:



La función lee el `2` y encuentra un salto de línea. El estado en el que queda el programa es éste:



Fíjate bien en qué ha ocurrido: nos hemos quedado a las puertas de procesar el salto de línea. Cuando el programa pasa a ejecutar el siguiente *gets*, ¡lee una cadena vacía! ¿Por qué? Porque *gets* lee caracteres hasta el primer salto de línea, y el primer carácter con que nos encontramos ya es un salto de línea. Pasamos, pues, a este nuevo estado:



¿Cómo podemos evitar este problema? Una solución posible consiste en consumir la cadena vacía con un *gets* extra y una variable auxiliar. Fíjate en este programa:

lee_alterno_bien.c

```
#include <stdio.h>

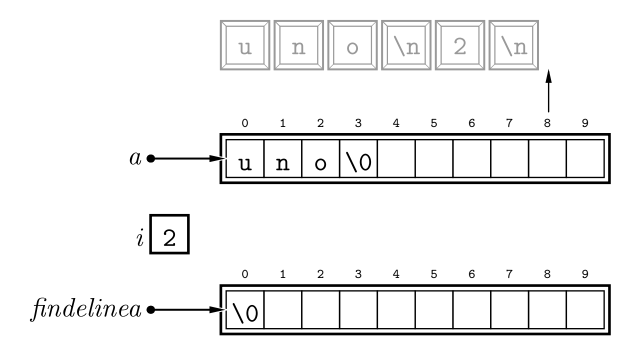
#define TALLA 80

int main(void)
{
  char a[TALLA+1], b[TALLA+1];
  int i;
  char findelinea[TALLA+1]; // Cadena auxiliar. Su contenido no nos importa.

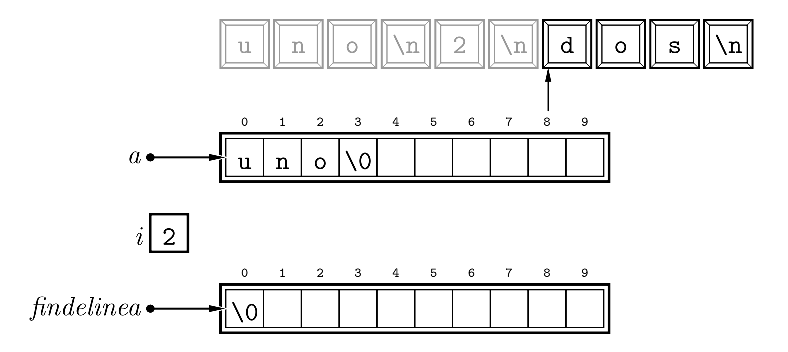
  printf("Cadena␣a:␣"); gets(a);
  printf("Entero␣i:␣"); scanf("%d", &i); gets(findelinea);
  printf("Cadena␣b:␣"); gets(b);
  printf ("La␣cadena␣a␣es␣%s,␣el␣entero␣i␣es␣%d␣y␣la␣cadena␣b␣es␣%s\n",
          a, i, b);

  return 0;
}
```

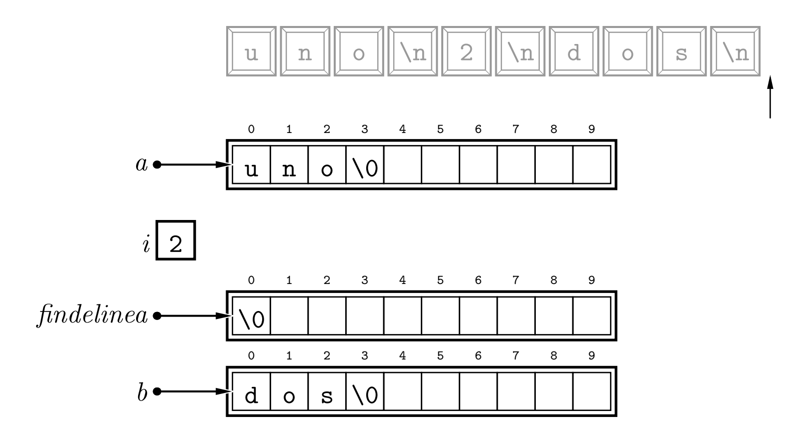
Hemos introducido una variable extra, *findelinea*, cuyo único objetivo es consumir lo que *scanf* no ha consumido. Gracias a ella, éste es el estado en que nos encontramos justo antes de empezar la lectura de *b*:



El usuario escribe entonces el texto que desea almacenar en b:



Ahora la lectura de `b` tiene éxito. Tras ejecutar *gets*, éste es el estado resultante:



¡Perfecto! Ya te dijimos que aprender C iba a suponer enfrentarse a algunas dificultades de carácter técnico. La única forma de superarlas es conocer bien qué ocurre en las entrañas del programa.

Pese a que esta solución funciona, facilita la comisión de errores. Hemos de recordar consumir el fin de línea sólo en ciertos contexto. Esta otra solución es más sistemática: leer siempre línea a línea con *gets* y, cuando hay de leerse un dato entero, flotante, etc., hacerlo con *sscanf* sobre la cadena leída:

lee_alterno_bien.c lee_alterno_bien.c

```
#include <stdio.h>

#define TALLA 80

int main(void)
{
  char a[TALLA+1], b[TALLA+1];
  int i;
  char linea[TALLA+1]; // Cadena auxiliar. Su contenido no nos importa.

  printf("Cadena␣a:␣"); gets(a);
  printf("Entero␣i:␣"); gets(linea); sscanf(linea, "%d", &i);
  printf("Cadena␣b:␣"); gets(b);
  printf ("La␣cadena␣a␣es␣%s,␣el␣entero␣i␣es␣%d␣y␣la␣cadena␣b␣es␣%s\n",
          a, i, b);

  return 0;
}
```

«¡Ah, ya sé!, ¡es un libro del Espejo, naturalmente! Si lo pongo delante de un espejo, las palabras se verán otra vez al derecho.»
Y éste es el poema que leyó Alicia[11]:

JERIGÓNDOR

Cocillaba el día y las tovas agilimosas
giroscopaban y barrenaban en el larde.
Todos debirables estaban los burgovos,
y silbramaban las alecas rastas.

11. [...] Carroll pasa a continuación a interpretar las palabras de la manera siguiente:

- BRYLLIG *["cocillaba"] (der. del verbo "*BRYL*" o "*BROIL*"); "hora de cocinar la comida; es decir, cerca de la hora de comer".*
- SLYTHY *["agilimosas"] (voz compuesta por "*SLIMY*" y "*LITHE*". "Suave y activo".*
- TOVA. *Especie de tejón. Tenía suave pelo blanco, largas patas traseras y cuernos cortos como de ciervo, se alimentaba principalmente de queso.*
- GYRE *["giroscopar"], verbo (derivado de* GYAOUR *o* GIAOUR*, "perro"). "Arañar como un perro".*
- GYMBLE *["barrenar"], (de donde viene* GIMBLET *["barrena"]) "hacer agujeros en algo".*
- WAVE *["larde"] (derivado del verbo "*to swab*" ["fregar"] o "soak" ["empapar"]). "Ladera de una colina" (del hecho de* empaparse *por acción de la lluvia).*
- MIMSY *(de donde viene* MIMSERABLE *y* MISERABLE*): "infeliz".*
- BOROGOVE *["burgovo"], especie extinguida de loro. Carecía de alas, tenía el pico hacia arriba, y anidaba bajo los relojes de sol: se alimentaba de ternera.*
- MOME *["aleca"] (de donde viene* SOLEMOME *y* SOLEMNE*). Grave.*
- RATH *["rasta"]. Especie de tortuga de tierra. Cabeza erecta, boca de tiburón, patas anteriores torcidas, de manera que el animal caminaba sobre sus rodillas; cuerpo liso de color verde; se alimentaba de golondrinas y ostras.*
- OUTGRABE *["silbramar"]. Pretérito del verbo* OUTGRIBE *(emparentado con el antiguo* TO GRIKE *o* SHRIKE*, del que proceden "*SHREAK*" ["chillar"] y "*CREAK*" ["chirriar"]: "chillaban".*

Por tanto, el pasaje dice literalmente: "Era por la tarde, y los tejones, suaves y activos, hurgaban y hacían agujeros en las laderas; los loros eran muy desdichados, y las graves tortugas proferían chillidos."

ALICIA ANOTADA (EDICIÓN DE MARTIN GARDNER), *Lewis Carroll.*